Self-stabilizing Population Protocols

Dana Angluin, James Aspnes, Michael J. Fischer
and
Hong Jiang
Yale University

This paper studies self-stabilization in networks of anonymous, asynchronously interacting nodes where the size of the network is unknown. Constant-space protocols are given for Dijkstra-style round-robin token circulation, leader election in rings, 2-hop coloring in degree-bounded graphs, and establishing consistent global orientation in an undirected ring. A protocol to construct a spanning tree in regular graphs using $O(\log D)$ memory is also given, where D is the diameter of the graph. A general method for eliminating nondeterministic transitions from the self-stabilizing implementation of a large family of behaviors is used to simplify the constructions, and general conditions under which protocol composition preserves behavior are used in proving their correctness.

Categories and Subject Descriptors: C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*; B.8.1 [**Performance**; **Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Anonymous, fairness, finite-state, population protocols, self-stabilization, sensor networks

1. INTRODUCTION

In some practical scenarios, a large and sometimes unknown number of devices are deployed over a region without fine control of their locations, communication and movement patterns. The devices are all indistinguishable and have only a few bits of memory each. Such scenarios are modeled by the *population protocols* introduced in [Angluin et al. 2006], where families of predicates computable in this model are explored. Graph properties computable in the same model are discussed in [Angluin et al. 2005]. Communication in population protocols occurs through pairwise interaction of anonymous finite-state nodes. The number of nodes is finite but unbounded. A communication graph describes which pairs of nodes

A preliminary version of the present work appeared in the proceedings of the *Ninth International Conference on Principles of Distributed Systems*, December 2005.
James Aspnes was supported in part by NSF grants CNS-0305258 and CNS-0435201.
Hong Jiang was supported in part by NSF grant ITR-0331548.
Corresponding author: James Aspnes, 51 Prospect Street, New Haven, CT 06520-8285, U. S. A., Email: james.aspnes@yale.edu, Telephone: +1-203-432-1232, Facsimile: +1-203-432-0593

may interact.

In the theoretical literature on distributed computing, some variant of a weak fairness condition is usually assumed. Informally, in an infinite fair execution each process or node satisfying certain conditions is given a turn infinitely often. We call this definition *local fairness*. The environment/scheduler is viewed as a powerful adversary who can strategically determine the sequence in which processes are activated, as long as local fairness is preserved. Many impossibility results rely on this assumption. For instance, the impossibility of deterministic self-stabilizing token circulation in uniform rings [Dijkstra 1974] follows from the assumption that the scheduler can activate the nodes in a round-robin fashion, preserving symmetry while achieving local fairness.

However, in practical distributed systems, such a powerful scheduler seldom exists. The global ordering of computational steps depends on a variety of elements. Temperature and power-supply affect the efficiency of electronic devices. Local clock frequency influences the progress of each node. For ad hoc networks, the movement of nodes determines possible sequences of interactions. Random delay is usually used in practical leader election and collision detection protocols, which can be viewed as a way to randomize the scheduling of the system.

In the model of population protocols, an alternative fairness condition called *global fairness* is assumed, which better reflects the scheduling properties of many distributed systems. Global fairness puts more constraints on the scheduler, so problems proved impossible under global fairness are also impossible under local fairness. Global fairness also provides a simple conceptual framework for protocol design. For instance, once a task is known to be possible in our model, randomization techniques can be applied to make the protocol work under some weaker fairness conditions.

The responsibility of many systems is to meet certain specifications for well-behavedness such as avoidance of deadlock, fairness among processes, fault tolerance, and other global system properties that cannot be simply modeled as functional computation. We extend the model of population protocols to accommodate such tasks. We focus on *self-stabilizing* systems that can start in any global configuration and achieve behavior meeting the task specification by itself. Such systems can tolerate worst-case transient faults.

In our model, nodes are finite-state and strongly anonymous, which means not only that they do not have unique IDs, but also that they do not have innate ability to determine whether two messages come from the same source. This condition well reflects the scenario of ad hoc networks consisting of identical nodes.

In Section 2, we give formal definitions of the new concepts that we use to extend the original model.

In Section 3, we present a powerful technique to facilitate the design and analysis of protocols by allowing the use of nondeterministic transitions. A large family of behaviors called *elastic behaviors* is defined that includes all behaviors we consider in this paper. We show that nondeterminism in protocols does not increase the class of elastic behaviors that have self-stabilizing implementations.

In Section 4, we introduce a framework for composing certain self-stabilizing protocols.

From Section 5 to Section 9, we give self-stabilizing protocols for various problems. As a simple example, we describe a self-stabilizing token circulation protocol in a ring with a pre-selected leader. The protocol resembles Dijkstra's "k-state" mutual-exclusion algorithm [Dijkstra 1974] but uses only $O(1)$ bits per node. We describe a self-stabilizing 2-hop coloring protocol which colors the nodes such that no node has two neighbors colored the same given a bound on the number of neighbors. Given the coloring protocol, we show how to direct an undirected ring. We also give a self-stabilizing protocol which constructs a spanning tree in regular graphs using $O(\log D)$ bits per node, where D is the diameter of the network. To remove the assumption of a pre-selected leader in some protocols, we construct a family of leader-election protocols, each corresponding to a class of rings. We also show that self-stabilizing leader election is generally impossible in our model without some restriction on topology.

In Section 10, we conclude with a brief summary of results and directions for further research.

1.1 Other Related Work

Self-stabilizing systems were first introduced by Dijkstra [Dijkstra 1974]. In his seminal paper, Dijkstra gives three protocols to achieve process mutual-exclusion in rings in a self-stabilizing way. Leader election and token management are fundamental problems in self-stabilization and have been extensively studied in various other models. Mayer, Ofek, Ostrovsky, and Yung [Mayer et al. 1992] give a constant-space probabilistic protocol for a self-stabilizing round-robin token management scheme on an anonymous bidirectional ring of identical processors, assuming an external timeout mechanism to detect deadlocks. Itkis, Lin, and Simon [Itkis et al. 1995] present a deterministic constant-space self-stabilizing protocol for leader election on uniform bidirectional asynchronous rings of prime size. In their model, there is a central daemon that picks an enabled processor each time to make an atomic move. The chosen processor can read the states of its two neighbors at the same time to determine its next state. Gouda and Haddix [Gouda and Haddix 1996] present a self-stablizing token circulation algorithm on unidirectional rings with one distinguished node. Each node in their algorithm has eight states. Higham and Myers [Higham and Myers 1999] give a randomized self-stabilizing algorithm that solves token circulation and leader election on anonymous, uniform, synchronous, and unidirectional rings of arbitrary but known size, in which each processor state and message has size in $O(\log n)$. Dolev, Israeli, and Moran [Dolev et al. 1997] present a self-stabilizing leader election protocol that tolerates addition or deletion of processors and links. Their protocol uses $O(\log n)$ bits per node. Beauquier, Gradinariu, and Johnen [Beauquier et al. 1999] present a silent and deterministic self-stabilizing leader-election protocol requiring constant memory space on unidirectional, ID-based rings where the ID values are bounded. A protocol is silent if from an arbitrary configuration the system reaches a configuration after which the communication registers of each node remain constant. They also prove a non-constant lower bound on space for self-stabilizing leader-election in unidirectional anonymous rings under a weak fairness assumption.

Herman [Herman 1990] proposes a probabilistic synchronous self-stabilizing token-circulation algorithm for identical nodes in an odd ring. Johnen [Johnen 2004] presents a randomized self-stabilizing token circulation protocol on unidirectional anonymous rings. Fairness is enforced by randomization and the fair circulation of *privileges*. The scheduler can only choose nodes that hold a privilege token to make the next step.

Itkis and Levin [Itkis and Levin 1994] present a self-stabilizing leader-election protocol for asynchronous networks of identical nameless nodes with arbitrary topology. In their model, each node's state consists of bits and pointers to immediate neighbors. The bits of a node x are visible to any neighbor y, and y can detect whether a pointer of x points to x or to y. Each node can detect a neighbor whose state satisfies a given property, set a pointer to it, and change state based on the above information. Our impossibility result for leader election in connected interaction graphs with arbitrary topology in Section 9 shows that their model and ours differ. However, it is an open problem whether our model can simulate theirs in some special classes of interaction graphs.

[Griggs and Yeh 1992] introduces the graph labeling problem with conditions at distance 2. The communication networks community has studied variants of this problem, for applications such as assigning radio frequency ranges or time slots to wireless-signal transmitters to avoid interference. Herman and Tixeuil [Herman and Tixeuil 2004] describe a randomized self-stabilizing coloring algorithm for wireless sensor networks. Although they consider ad hoc networks and do not assume each node has prior knowledge of its neighborhood, the nodes have unique identifiers. Under the assumption of a known upper bound of neighborhood size, their algorithm uses a small set of colors to label the communication graph. This is a generalization of the renaming problem [Attiya et al. 1987]. Gradinariu and Johnen [Gradinariu and Johnen 2001] give a probabilistic solution for neighborhood unique naming in anonymous networks. No explicit identifiers are assumed in the model, however, each node has the ability to associate incoming messages with neighbors, possibly by physical communication links. Moscibroda and Wattenhofer [Moscibroda and Wattenhofer 2005] present another randomized coloring algorithm for radio networks. Their algorithm has the property that nodes can join the network asynchronously. The algorithm is not self-stabilizing and also assumes the existence of unique identifiers to let a receiver recognize whether or not two different messages are sent by the same sender. We study a similar 2-hop coloring problem which requires distinct neighbors of each node receive different colors, but two adjacent nodes can have the same color as long as they do not share any neighbor. We impose a strong anonymity condition: A node does not have the innate ability to distinguish different neighbors. To our knowledge no existing protocol is applicable in this model. In Section 6, we give a self-stabilizing 2-hop coloring protocol given a known degree bound. A direct application is to assign local identifiers so that nodes are able to refer to specific neighbors, whereas such an ability is implicitly assumed in the above mentioned coloring algorithms.

2. BASIC MODEL

In the population protocols model, a network is represented by a directed graph $G = (V, E)$ with n vertices numbered 0 through $n - 1$ and no multi-edges or self-loops. In this paper, all network graphs are assumed to be weakly connected.

Each vertex represents a finite-state sensing device, and an edge (u, v) indicates the possibility of a communication between u and v in which u is the *initiator* and v is the *responder*. The distinct roles of the two devices in an interaction is a fundamental assumption in our model. An "undirected" communication graph refers to a network in which for every edge (u, v), interactions of the forms (u, v) and (v, u) are both possible.

When presenting protocols in the population protocols model in the rest of this paper, we abbreviate "population protocol" to "protocol" in order to avoid verbosity. Readers who are interested in the relationship between population protocols and more traditional message-passing protocols could refer to [Angluin et al. 2007] for a detailed treatment.

A *protocol* $P(Q, \mathcal{C}, X, Y, O, \delta)$ consists of a finite set of *states* Q, a set of *initial configurations* $\mathcal{C}$, a finite set X of *input symbols*, an *output function* $O : Q \to Y$, where Y is a finite set of *output symbols*, and a *transition function* δ mapping each element of $(Q \times X) \times (Q \times X)$ to a nonempty subset of $Q \times Q$. If $(p', q') \in \delta((p, x), (q, y))$, we call $((p, x), (q, y)) \to (p', q')$ a *transition*. The transition function, and the protocol, is *deterministic* if $\delta((p, x), (q, y))$ always contains just one pair of states. The inputs provide a way for a protocol to interact with an external entity, be it the environment, a user, or another protocol. If X is empty, the protocol does not accept input.

A *configuration* is a mapping $C : V \to Q$ and an *input assignment* is a mapping $\alpha : V \to X$. The state of each device in the network is given by C and the input is given by α. If X is empty, α is also empty, and C specifies a configuration. A *trace* $T_G(Z)$ on a graph $G(V, E)$ is an infinite sequence of assignments from V to the symbol set Z: $T_G = \lambda_0, \lambda_1, \ldots$ where λ_i is an assignment from V to Z. Z is called the alphabet of T_G. If $Z = X$, we say T_G is an *input trace* of the protocol. A trace is called a constant trace if it is a sequence of the same assignment. Let C and C' be configurations, α be an input assignment, and u, v be distinct nodes. We say that (C, α) goes to C' via pair $e = (u, v)$, denoted $(C, \alpha) \xrightarrow{e} C'$, if the pair $(C'(u), C'(v))$ is in $\delta((C(u), \alpha(u)), (C(v), \alpha(v)))$ and for all $w \in V - \{u, v\}$ we have $C'(w) = C(w)$. We say that (C, α) can go to C' in one step, denoted $(C, \alpha) \to C'$, if $(C, \alpha) \xrightarrow{e} C'$ for some edge $e \in E$. If α is empty, we simply write $C \xrightarrow{e} C'$ and $C \to C'$ respectively. Given an input trace $IT = \alpha_0, \alpha_1, \ldots$ we write $C \xrightarrow{*} C'$ if there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $(C_i, \alpha_i) \to C_{i+1}$ for all i, $0 \leq i < k$, in which case we say that C' is *reachable* from C given input trace IT. The same notation is used for protocols that do not accept inputs.

An *execution* is an infinite sequence of configurations and input assignments (C_0, α_0), (C_1, α_1), …such that $C_0 \in \mathcal{C}$ and for each i, $(C_i, \alpha_i) \to C_{i+1}$. We extend the output function O to take a configuration C and produce an *output assignment* $O(C)$ defined by $O(C)(v) = O(C(v))$. Let $\sigma = (C_0,\ \alpha_0), (C_1, \alpha_1), \ldots, (C_i, \alpha_i), \ldots$ be an execution of P. We define the *output trace* of an execution as $OT(\sigma) = O(C_0), O(C_1), \ldots, O(C_i), \ldots$.

Unlike most models in the literature, in population protocols fairness is a global property, therefore we call this fairness condition *global fairness*.

Definition 2.1 *Global Fairness.* An execution is *fair* if for every α, C and C'

such that $(C, \alpha) \to C'$, if (C, α) occurs infinitely often in the execution, then C' also occurs infinitely often in the execution.

An execution is generated by an infinite sequence of interactions. Intuitively, if the interactions in this sequence are chosen by an adversarial scheduler in accordance with the global fairness condition, the scheduler can not prevent an interaction from occurring infinitely often in a given configuration and input assignment unless that configuration-input pair occurs only finitely many times.

Our definitions are actually slightly weaker. Our notion of execution specifies only the resulting configuration of each transition, not the actual transition that causes that result. In case two different transitions lead to the same result (as can happen for example with "no-op" transitions), the execution only shows the resulting configuration, not which transition led to that result. Similarly, our formal definition of global fairness does not insist that a particular transition be activated in a certain configuration-input pair; it requires only that the configuration that would result, were the transition to be activated, must eventually occur. Note that we do not require that it ever occur as the very next configuration.

One important use of global fairness is to show that if a certain execution fragment is possible from a given infinitely-occurring configuration, then the configuration that would result from executing that fragment must also occur infinitely often in the execution, subject to some contraints on the inputs.

THEOREM 2.2. *Let σ be an execution that satisfies global fairness, and assume C_0 occurs infinitely often in σ. Let $\sigma' = (C_0, \alpha_0), (C_1, \alpha_1), \ldots, (C_k, \alpha_k)$ be an execution fragment. Suppose the α_i's are such that if C_i occurs infinitely often in σ, then (C_i, α_i) occurs infinitely often in σ $(0 \leq i < k)$. Then C_k occurs infinitely often in σ.*

PROOF. Global fairness implies that if (C_{k-1}, α_{k-1}) occurs infinitely often in σ, then C_k occurs infinitely often in σ. The result follows by an easy induction on k. □

A self-stabilizing system can start at an arbitrary configuration and eventually exhibit "good" behavior. We define a *behavior* B on a network $G(V, E)$ to be a set of traces on G that have the same alphabet. We write $B(Z)$ to be explicit about the common alphabet Z. A behavior B is *constant* if every trace in B is constant. If given the constraint that every input trace is contained in some behavior $B_{\text{in}}(X)$, the output trace of every fair execution of a protocol $P(Q, \mathcal{C}, X, Y, O, \delta)$ starting from any configuration in $\mathcal{C}$ is in some behavior $B_{\text{out}}(Y)$, we say P is an *implementation* of output behavior B_{out} given input behavior B_{in}. If P does not have any restriction on inputs, we simply say P is an implementation of B_{out}. Given a behavior $B(Z)$, we define the corresponding *stable behavior* $B^s(Z) = Z^* B(Z)$. Thus, an execution in a stable behavior has a completely arbitrary finite prefix followed by an execution with the desired properties. If $P(Q, \mathcal{C}, X, Y, O, \delta)$ is an implementation of B^s, and $\mathcal{C}$ is the set of all possible configurations, we say that P is a *self-stabilizing implementation* of B.

3. NONDETERMINISTIC PROTOCOLS

In section 2, we gave the definition of deterministic protocol and nondeterministic protocol. Nondeterminism in this model is different from nondeterminism in computation theory. A nondeterministic Turing machine is allowed to always guess the "correct" choice of multiple possible steps. In the execution of a population protocol, the choice is made by an adversarial scheduler who is only limited by global fairness.

We define the *repetition closure* of a sequence t to be the set of sequences obtainable from t by repeating each element one or more times. In other words, given any sequence $t = a_1 a_2 \dots a_i \dots$, the repetition closure $R(t)$ is $a_1^+ a_2^+ \dots a_i^+ \dots$ in regular expression notation. We extend the definition of R to a behavior B by taking the union of $R(t)$ for all $t \in B$. We say a behavior B is *elastic* if $B = R(B)$. We show below that for elastic behaviors nondeterministic protocols are not more powerful than deterministic ones, in the sense that if there exists a nondeterministic self-stabilizing implementation of an elastic behavior, there also exists a deterministic version. We only consider networks of at least three nodes.

We construct a compiler to convert every nondeterministic protocol to a deterministic one. To preserve self-stabilization, the compiler itself must be self-stabilizing.

Let P_1 be a nondeterministic protocol with states Q, input alphabet X, and transition function δ. We describe a simulation of P_1 that works in graphs with at least three vertices. Let m be the maximum cardinality of any of the sets $\delta((q, x), (q', x'))$ for $q, q' \in Q$ and $x, x' \in X$. For each $q, q' \in Q$ and $x, x' \in X$, select an arbitrary surjective function $f_{(q,x),(q',x')}$ mapping $\{0, 1, \dots, m-1\}$ to $\delta((q, x), (q', x'))$.

We describe a protocol P_2 to simulate each step of P_1 by multiple deterministic steps. The state components used in P_1 only change in the last of the corresponding steps in P_2. The state consists of three components:

(1) a state $q \in Q$,

(2) a mark $\bullet$, which we call a nondeterminizer token (abbreviated as token from here on for simplicity), or its absence $\circ$,

(3) a choice counter, consisting of an integer between 0 and $m - 1$ inclusive.

There are four types of transition rules in P_2:

$$\begin{array}{lll}
\text{Rule 1.} & (([q \bullet c], x), ([q' \bullet c'], x')) \to ([q \circ c], & [q' \bullet c']) \\
\text{Rule 2.} & (([q \bullet c], x), ([q' \circ c'], x')) \to ([q \circ c], & [q' \bullet (c' + 1)]) \\
\text{Rule 3.} & (([q \circ c], x), ([q' \bullet c'], x')) \to ([q \bullet (c + 1)], & [q' \circ c']) \\
\text{Rule 4.} & (([q \circ c], x), ([q' \circ c'], x')) \to ([r \circ c], & [r' \bullet c'])
\end{array}$$

where the increments are made modulo m and $(r, r') = f_{(q,x),(q',x')}(c)$ is the pair of states in $\delta((q, x), (q', x'))$. Thus, in transition rule 4, the value of the choice counter of the initiator is used to make a deterministic choice of an element of $\delta((q, x), (q', x'))$. The role of the tokens is to hop around the graph incrementing choice counters as they go. The first three transition rules accomplish this purpose. Transition rule 4 ensure that deadlock is impossible: even if we start from a configuration with no tokens, the rule will generate new tokens. Transition rule 1 ensure that the tokens have room to move around by merging two adjacent tokens.

We call the shared state component of P_1 and P_2 *the q component.* If C_1 is a configuration in P_1, and C_2 is a configuration in P_2, we say $C_1 \lhd C_2$ if each node in C_1 has the same q component as the corresponding node in C_2.

LEMMA 3.1. *Assume a connected network with $n \geq 2$ vertices. Starting from an arbitrary configuration, every globally fair execution of P_2 eventually reaches a point after which the number of tokens in every configuration is between 1 and $n-1$.*

PROOF. Assume the initial configuration contains at least one token. If there are n tokens, the next interaction will remove one of them, because only the first transition rule is applicable. Starting from a configuration with at most $n-1$ tokens, it is impossible to generate the nth token, because the transitions that generates new tokens (rule 4) is applicable only when two nodes without the token interact. It is also impossible for all the tokens to disappear, because the number of tokens only decrease when two merge into one by applying transitions of rule 1.

If the initial configuration does not contain any token, the next interaction will generate one, because only rule 4 is applicable.

Therefore, after the first step, there will be at least 1 and at most $n-1$ tokens. It is never possible thereafter to generate the nth token, nor is it possible for all tokens to disappear. □

LEMMA 3.2. *Assume a connected network with $n \geq 2$ vertices. Starting from a configuration with at least one and at most $n-1$ tokens, a token can be generated at any node without changing the q component of any node.*

PROOF. The only transitions that change the q component are of rule 4 which require that both the initiator and the responder do not have a token. The token can be moved to any node by applying the other transition rules. In particular, transition rule 2 and rule 3 move a token in either direction on an arbitrary edge with an existing token on one end and no token on the other. □

LEMMA 3.3. *In a connected network of at least three nodes, for any possible step $(C_1, \alpha) \to C_1'$ of P_1 and any C_2 such that there is at least one and at most $n-1$ tokens in C_2 and $C_1 \lhd C_2$, where n is the number of nodes, there exists C_2' such that $C_1' \lhd C_2'$ and an admissible sequence of steps of P_2 from (C_2, α) to C_2' given the constant input trace with input assignment α.*

PROOF. Let (u, v) be the activated edge, and $((q_1, x_1), (q_2, x_2)) \to (r_1, r_2)$ be the transition in P_1 corresponding to the step $(C_1, \alpha) \to C_1'$. We show there exists a sequence of steps in P_2 that starts from (C_2, α) and reaches some C_2' where $C_2 \lhd C_2'$.

We construct a sequence of steps using rules 1–3 to reach a configuration where the states of u and v are $[q_1 \circ c_1]$ and $[q_2 \circ c_2]$ respectively, $f_{(q_1,x_1),(q_2,x_2)}(c_1) = (r_1, r_2)$, and no q component of any state has changed. A final activation of (u, v) results in the desired configuration C_2'.

First, we apply the sequence guaranteed by Lemma 3.2 to place a token at v without changing the q component of any node. If there is now a token at u, activate (u, v) twice; otherwise activate (u, v) once. Now there is one token at u and no token at v.

Next, activate (u, v) repeatedly. The token moves back and forth between u and v, and c_1 cycles through all possible values. Stop when c_1 assumes a value such that $f_{(q_1,x_1),(q_2,x_2)}(c_1) = (r_1, r_2)$. Now the token is at u.

Next, activate edges without changing the q component of any node so as to leave both u and v without tokens. There are two cases. (1) If u is adjacent to another node w ($w \neq v$), we remove the token from u by repeatedly activating the edge connecting u and w. The number of activations depends on the direction of the edge and whether w also has a token. (2) If v is the only neighbor of u, then v is adjacent to another node w ($w \neq v$) since we assume the network has at least three nodes and is connected. Activate (u, v) once to move the token to v. We remove the token from v by repeatedly activating the edge connecting v and w. Again, the number of interactions depends on the direction of the edge and whether w also has a token.

Now, neither u nor v has a token. Activate edge (u, v). Rule 4 applies, and the resulting configuration C_2' has identical q components with C_1'; hence, $C_1' \lhd C_2'$. □

Lemma 3.4. *For any step $(C_2, \alpha) \to C_2'$ in an execution of P_2 in which the q component of some node is changed, there exist configurations C_1 and C_1' of P_1 such that $C_1 \lhd C_2$, $C_1' \lhd C_2'$, and $(C_1, \alpha) \to C_1'$ is a transition of P_1.*

Proof. The only transitions in P_2 that change the q component of some node are applications of rule 4 which by definition is derived from the transitions in P_1. Therefore for every possible step that applies rule 4, there exists a corresponding possible step in P_1. □

Theorem 3.5. *If a nondeterministic protocol P_1 is a self-stabilizing implementation of a behavior B on a connected network of at least three vertices given a constant behavior B', there exists a deterministic protocol P_2 that is a self-stabilizing implementation of $R(B)$ given B'.*

Proof. Let P_2 be the protocol from our construction. Let $\Pi_q : [qmc] \mapsto q$ be the projection that maps states of P_2 to states of P_1 by erasing all but the q component. Extend Π_q in the natural way to map configurations, sets of configurations, and executions of P_2 to the corresponding objects of P_1. Let E_2 be an execution of P_2. From Lemma 3.4, $\Pi_q(E_2)$ is contained in the repetition closure $R(E_1)$ for some execution E_1 of P_1.

Let $\mathcal{A}$ be the set of all possible configurations of P_2, and let $\mathcal{B} \subset \mathcal{A}$ be the set of all configurations in which the number of tokens is between 1 and $n - 1$, where n is the number of nodes. Obviously, $\Pi_q(\mathcal{A}) = \Pi_q(\mathcal{B})$. From Lemma 3.4, for any execution E_2 of P_2 starting from a configuration in $\mathcal{B}$, there exists an execution E_1 of P_1, such that $\Pi_q(E_2) \in R(E_1)$. From Lemma 3.1, every fair execution of P_2 has a suffix in which every configuration is in $\mathcal{B}$.

The above shows that for any fair execution E_2 of P_2, there exists an execution E_1 of P_1 such that $\Pi_q(E_2)$ has a suffix in $R(E_1)$. Next, we show that E_1 is a fair execution of P_1.

Let C_1 be a configuration of P_1, $(C_1, \alpha) \to C_1'$ a transition of P_1, and suppose C_1 occurs infinitely often in E_1. There is some configuration C_2 with $C_1 \lhd C_2$

that occurs infinitely often in E_2. From Lemma 3.3, there exists a configuration C_2' of P_2 such that $C_1' \lhd C_2'$ and C_2' can be reached from (C_2, α) by an admissible sequence of steps of P_2 given the constant input trace with input assignment α. By Theorem 2.2, since E_2 is fair, C_2' occurs infinitely often in E_2. Then C_1' occurs infinitely often in E_1. Thus, E_1 satisfies global fairness, as desired.

Theorem 3.5 immediately follows from the definition of behavior and that the output of each node is determined by its state. □

If B is an elastic behavior, $B = R(B)$. The following corollary is immediate:

COROLLARY 3.6. *If a nondeterministic protocol P_1 is a self-stabilizing implementation of an elastic behavior B given a constant behavior B', there exists a deterministic protocol P_2 that is a self-stabilizing implementation of B given B'.*

4. PROTOCOL COMPOSITION

In complex protocols, sometimes it is desirable to utilize existing protocols as modules. Parallel execution of protocols is easily achieved by taking the Cartesian product of their state sets and updating the states for each protocol independently when a transition occurs. In this section we introduce one technique of protocol composition in our model and show how to compose protocols P_1, P_2, ..., P_n, with some restrictions.

For $n = 2$, assuming P_1 and P_2 access different components of the node's state, we run P_1 and P_2 in parallel, except that whenever P_2 is executed, it uses the current output of P_1 as its current input. When an edge is fired, it is nondeterministically determined which protocol gets the chance to execute. Recall that a behavior is constant if it contains only constant traces.

THEOREM 4.1. *Suppose B_1 is a constant behavior. If P_2 is a self-stabilizing implementation of an elastic behavior B_2 given input behavior B_1, and P_1 is a self-stabilizing implementation of B_1, the composition of P_1 and P_2 (written as $P_1 \circ P_2$) is a self-stabilizing implementation of B_2.*

PROOF. Let $S = C_0, C_1, \ldots$ be any fair execution of $P_1 \circ P_2$. Define the projection $\Pi_1(C)$ of a configuration C to be the sub-configuration produced by taking each node's state components that are accessed by P_1. Π_2 is defined similarly for P_2. Define $S' = C_0', C_1', \ldots$ the maximal subsequence of S in which for each i, the transition immediately after C_i' is defined in P_1. $S'' = C_0'', C_1'', \ldots$ is defined similarly for P_2. Because P_1 is self-stabilizing, and $\Pi_1(C_0'), \Pi_1(C_1'), \ldots$ is a fair execution of P_1, there exists some i such that the output trace of $\Pi_1(C_i'), \Pi_1(C_{i+1}'), \ldots$ satisfies B_1. Let C_j'' be any configuration that appears after C_i' in S, and let $C_j'', C_{j+1}'', \ldots$ be the sequence starting from C_j'' in S''. Because the output trace of P_1 in $C_j'', C_{j+1}'', \ldots$ is constant and satisfies B_1, $\Pi_2(C_j''), \Pi_2(C_{j+1}''), \ldots$ is a fair execution of P_2 whose output trace satisfies B_2. Because B_2 is elastic, the output trace of $P_1 \circ P_2$ in the subsequence of S starting from C_j'' satisfies B_2. Therefore $P_1 \circ P_2$ is a self-stabilizing implementation of B_2. □

5. TOKEN-CIRCULATION IN AN ORIENTED RING

As a simple example, we discuss the token circulation problem in an interaction graph whose topology is an oriented ring. The protocol uses the same idea as in Dijkstra's first algorithm in [Dijkstra 1974], but we only use 2 colors (0 and 1). Readers from the self-stabilization community will find the protocol familiar.

Definition 5.1. The token-circulation behavior TC on graph $G(V, E)$ is the set of all traces $t = \beta_0, \beta_1, \ldots$ with alphabet $\{T, \phi\}$ such that:

(1) For all $m \geq 0$, $\exists v \in V$ such that $\beta_m(v) = T$ and $\forall u \in V - \{v\}$, $\beta_m(u) = \phi$.

(2) For all $0 \leq m < k < n$, if $\exists v, w \in V$ $(v \neq w)$ such that $\beta_m(v) = \beta_k(w) = \beta_n(v) = T$, then $\forall u \in V - \{v\}$, $\exists l$ such that $m < l < n$ and $\beta_l(u) = T$.

(3) For all $v \in V$, $\beta_k(v) = T$ for infinitely many k.

A node owns a token in a configuration if its output is T. For any trace in TC, exactly one node has a token in each configuration, and after a node releases a token, it does not obtain a token again until every other node has obtained a token once.

We describe a self-stabilizing implementation of TC given the *leader-election behavior* LE. The description of a self-stabilizing implementation of LE is postponed to Section 9.

Definition 5.2. The leader-election behavior LE on graph $G = (V, E)$ is the set of all constant traces $\beta, \beta, \ldots$ such that for some $v \in V$, $\beta(v) = L$ and for all $u \neq v$, $\beta(u) = N$.

Informally, there is a static node with the leader mark L, and all other nodes have the nonleader mark N in every configuration. Given the LE input behavior, one node receives input L and all other nodes receive input N.

When two nodes interact, if the responder is the leader, it sets its label to the complement of the initiator's label; otherwise the responder copies the label from the initiator. If an interaction triggers a label change, a token is passed from the initiator to the responder. If a token is not present at the initiator, a new token is generated. Protocol 1 describes the process formally.

Protocol 1 Self-stabilizing token circulation in rings

Node states are pairs in $\{\circ, \bullet\} \times \{0, 1\}$. "$\bullet$" indicates the presence of a token and "$\circ$" indicates the absence of a token. The second component of a node is called the *label* of that node. The interaction rules are:

$$\begin{array}{ll} \text{Rule 1.} & ((*b, N), (*b, L)) \rightarrow (\circ b, \ \bullet \overline{b}) \\ \text{Rule 2.} & ((*b, *), (*\overline{b}, N)) \rightarrow (\circ b, \ \bullet b) \end{array}$$

We use the convention that $*$ on the left side of a rule matches any value for the component, that b on the left side matches either 0 or 1, and $\overline{b}$ means the complement of b. The output rules are $\bullet * \rightarrow T$ and $\circ * \rightarrow \phi$. (Output T if and only if the first component is "$\bullet$".)

Because each transition is independent of the presence or absence of tokens before the interaction, the token component of the state is only needed for production of the output symbol. In models with other conventions for specifying the location of the token, this extra state bit might not be needed.

Generally, proving self-stabilization in our model involves proving two statements: there exist some *legitimate configurations*, beginning with which every possible execution has a suffix satisfying the desired behavior, and a legitimate configuration is reachable from any arbitrary configuration. The fairness condition guarantees that every fair execution eventually reaches a legitimate configuration.

For Protocol 1, legitimate configurations are those in which every node has the same label. We establish the correctness of the protocol in the following lemmas:

LEMMA 5.3. *Let C_0 be a configuration in which every node has the same label. Given the input behavior LE, the trace of every fair execution of Protocol 1 starting with C_0 has a suffix in TC.*

PROOF. We define the direction of the edges to be “clockwise” and the opposite direction to be “counter-clockwise”. We call the leader node 0, and the sequence of nodes to the clockwise direction is 0, 1, ..., $n-1$. We use (u, v) to refer to the directed edge that starts at u pointing to v. The only edge in C_0 that will cause state change is $(n-1, 0)$ (by rule 1). When the edge is activated, node 0 obtains a token and is assigned a label that is different from all other nodes, and if node $n-1$ owns a token before the interaction, it surrenders the token. Suppose C_1 is the configuration immediately following C_0. Node 0's label cannot change until node $n-1$ has the same label, and every other node u cannot change its label until the label of node $u-1$ is different from its own. Therefore u is the only node that could change its label (when edge $(0, 1)$ is activated). If C_2 is the configuration immediately following C_1, it is easy to see that only $(1, 2)$ will cause state change. In general, for any u $(1 < u \leq n)$, the following is true in C_u:

(1) $u-1$ owns a token.

(2) For all $v < u-1$, v does not own a token.

(3) The only edge that can cause state change is $(u-1, u)$.

Note that in C_n every node again has the same label, and node $n-1$ is the only node that owns a token. Thus for $u \geq n$, in C_u, the only edge that can cause state change is $(u-1 \bmod n, u \bmod n)$, and node $u-1 \bmod n$ owns the unique token in the ring. □

LEMMA 5.4. *C_0 is reachable from any arbitrary configuration.*

PROOF. We use the same naming convention as in the proof of Lemma 5.3. Let the scheduler activate $(0, 1)$, $(1, 2)$, ..., $(n-2, n-1)$ sequentially. An activation of $(0, 1)$ can cause state change only when node 0 and node 1 have different labels, and the activation causes 1 to change its label to be the same as 0's. In general, for $0 \leq u < n-1$, the activation of $(u, u+1)$ causes node $u+1$ to copy the label of u if u and $u+1$ don't already have the same label. After $(u, u+1)$ is activated, 0, 1, ..., $u+1$ have the same label. Therefore, after $(n-2, n-1)$ is activated, nodes 0, 1, ..., $n-1$ all have the same label. □

THEOREM 5.5. *Protocol 1 is a constant-space self-stabilizing implementation of output behavior TC given input behavior LE in an oriented ring.*

Theorem 5.5 follows from Lemma 5.3 and Lemma 5.4

6. 2-HOP COLORING IN BOUNDED-DEGREE GRAPHS

To extend the token circulation algorithm to undirected rings, we need a protocol that imposes orientation on the ring. A necessary condition is that each node be able to recognize its two different neighbors. Here we describe a more general algorithm that enables each node in a degree-bounded graph to distinguish between its neighbors.

Suppose an undirected graph has degree bound d. We want to color the graph such that any two nodes adjacent to the same node have different colors. After a graph is properly colored, the neighbors of any node have different colors and thus are distinguishable. Notice that two nodes directly connected by an edge can have the same color if they are not both adjacent to the same node.

More precisely, for each node v, if u and w are distinct neighbors of v, then u and w must receive different colors. We call such a pair (u, w) a *2-hop pair*, and we call a node-coloring in which every 2-hop pair is distinctly colored a *2-hop coloring.* It is not difficult to see that $g = d(d-1)+1$ colors suffice, for the related graph whose edges are the 2-hop pairs of the original graph has degree bound $d' = d(d-1)$, and any graph of degree bound d' can be colored (in the ordinary sense) with at most $d' + 1$ colors.

Definition 6.1. The *2-hop coloring behavior* $2HC$ on graph $G = (V, E)$ with color set Γ is the set of constant traces $\lambda, \lambda, \ldots$ where the alphabet of λ is Γ and whenever $u, v, w \in V$ are such that $(u, v) \in E$ and $(v, w) \in E$ and $u \neq w$, we have $\lambda(u) \neq \lambda(w)$.

We give two protocols for solving 2-hop coloring for a graph with degree bound d using g colors. The first is presented as a nondeterministic protocol, from which the nondeterminism can be removed by applying Corollary 3.6. The second is a deterministic protocol that incorporates a simplified nondeterminizer sufficient to solve the problem at hand.

6.1 Nondeterministic protocol

In this and the following sections, we describe our algorithms by specifying the interaction between two adjacent nodes u and v when the edge (u, v) is activated. The pseudocode shows how the state variables of both u and v are updated in an interaction as if the state variables of both nodes lived in a common memory. This can easily be translated into our formal model whereby each node updates its own state variables based on their current values and the values of the other node's state variables, which are supplied to it as inputs during the interaction.

In Protocol 2, each node u has the following state components:

$color_u$	An integer encoding the color of node u. Its value is between 0 and $g-1$.
F_u	A bit array whose size is g, indexed by colors.

Node u outputs the current value of its $color_u$ component.

Protocol 2 Nondeterministic 2-hop coloring with degree bound d

For each node u:

State variables: $color_u$ and F_u

Output: $color_u$

The interaction between an initiator u and a responder v:

```
1: if $F_u[color_v] \neq F_v[color_u]$ then      ▷ possibly conflicting colors
2:     $color_u \leftarrow color'_u$               ▷ nondeterministic coloring
3:     $F_u[color_v] \leftarrow F_v[color_u]$
4: else                                          ▷ valid coloring
5:     $F_u[color_v] \leftarrow \overline{F_u[color_v]}$
6:     $F_v[color_u] \leftarrow \overline{F_v[color_u]}$
7: end if
```

The statement $color_u \leftarrow color'_u$ means one of the g possible colors is nondeterministically assigned as the new color of u. In other words, in this case there are g rules that can be applied to (u, v), each assigning a distinct color to u.

Distinct nodes u and w are *violating nodes* if they have the same color and share a common neighbor. Formally, they are *violating* if $color_u = color_w$ and there are edges (u, v) and (v, w) for some v. We also apply the term *violating* to a single node u if there exists a node w that it is in violation with.

Let (u, v) be an edge in the network. We say that the edge is *synchronized* if $F_u[color_v] = F_v[color_u]$. We also apply the term *synchronized* to the pair of nodes u and v, and we refer to the complementary case as *unsynchronized.*

A configuration is *legitimate* if it satisfies two safety conditions:

(1) *[No color violations]* There are no violating nodes.

(2) *[Synchronization]* There are no unsynchronized edges.

To get an intuition for the protocol, consider a legitimate configuration, and suppose network edge (u, v) is activated. The edge is synchronized, so lines 5 and 6 of the protocol are are executed. These do not change the color of any node. Although they do flip the bits $F_u[color_v]$ and $F_v[color_u]$, all edges remain synchronized. Clearly (u, v) is still synchronized, as is any edge (x, y) that does not share a node with (u, v). The other edges that touch u and v also remain synchronized because of the first safety condition. If (u, w) is such an edge, then $color_v \neq color_w$, so $F_u[color_w]$ is not affected by the change to $F_u[color_v]$, and (u, w) remains synchronized. Similarly, each edge (v, w) with $w \neq u$ remains synchronized. Hence, the new configuration is legitimate.

Now suppose the original configuration is not legitimate, (u, v) and (v, w) are network edges, and u and w have the same color c. Suppose edge (u, v) is activated. If it was synchronized before the activation, line 6 is executed and bit $F_v[c]$ is flipped. This causes v's synchronization status with w to change. In particular, if (v, w) was synchronized before the u-v interaction, then it loses its synchronization as a result. On the other hand, if (u, v) was not synchronized before the activation, then line 2 is executed and u is non-deterministically recolored, possibly causing it to become violating. In either case, though, the edge (u, v) is synchronized in the resulting configuration. We will show that after enough nondeterministic recolorings, the protocol will eventually reach a legitimate configuration.

We now make these arguments more precise.

LEMMA 6.2. *The trace of any execution of Protocol 2 starting from a legitimate configuration is in $2HC$.*

PROOF. Let C_0 be a legitimate configuration and (u,v) be any edge. Suppose $C_0 \xrightarrow{(u,v)} C_1$. Because (u,v) is synchronized in C_0, no color change occurs, and both $F_u[\mathit{color}_v]$ and $F_v[\mathit{color}_u]$ are complemented in the interaction. Hence, C_0 and C_1 have the same coloring, and (u,v) is synchronized in C_1. Since C_1 has no color violations, v is the only neighbor of u with color color_v, and u is the only neighbor of v with color_u. Therefore, the bits $F_u[\mathit{color}_v]$ and $F_v[\mathit{color}_u]$ which differ between C_0 and C_1 do not affect the synchronization status of any other edges, which remain synchronized in C_1. Thus, C_1 is legitimate and has the same coloring as C_0. Because C_0 and (u,v) are chosen arbitrarily, we conclude that every configuration in any execution starting from any legitimate configuration is legitimate, and all configurations in any such execution have the same coloring. It follows that the sequence of outputs is constant, and the trace is in $2HC$. □

LEMMA 6.3. *Starting from an arbitrary configuration, there exists a finite execution fragment of Protocol 2 that reaches a legitimate configuration.*

PROOF. We describe a two-phase procedure to construct a path to a legitimate configuration from an arbitrary configuration. The first phase corrects color violations, and the second phase synchronizes all network edges without changing node colors.

Phase 1 consists of a sequence of subphases $\sigma(u)$, one for each node u. Each subphase consist of a sequence of interactions as described below. The subphases are constructed sequentially and refer to the configuration that results from the previous subphases.

Subphase $\sigma(u)$ is empty if u is not violating at the start of the subphase, so assume now that u is violating. We construct a sequence of activations $\sigma(u)$ that change u's color so that it is no longer violating but do not change the color of any other node.

Since u is violating, there are edges (u,v) and (v,w) such that $\mathit{color}_u = \mathit{color}_w$. $\sigma(u)$ consists of the activations described by the cases below, which are based on the values of $F_u[\mathit{color}_v]$, $F_v[\mathit{color}_u] = F_v[\mathit{color}_w]$, and $F_w[\mathit{color}_v]$:

(1) $F_u[\mathit{color}_v] \neq F_v[\mathit{color}_u]$. Let the scheduler activate (u,v), so that line 2 is executed. Because u can be the endpoint of at most $d(d-1)$ 2-hop paths, there exists at least one color color'_u that differs from the color of every 2-hop neighbor of u. Let u choose color'_u as its new color. This removes the color violation from u. Line 3 sets $F_u[\mathit{color}_v] \leftarrow F_v[\mathit{color}'_u]$, so u and v are synchronized after the interaction.

(2) $F_u[\mathit{color}_v] = F_v[\mathit{color}_u] = F_w[\mathit{color}_v]$. Let the scheduler activate (v,w). This will flip $F_v[\mathit{color}_w]$ and $F_w[\mathit{color}_v]$, causing (u,v) to become unsynchronized. Now continue with case 1.

(3) $F_u[color_v] = F_v[color_u] \neq F_w[color_v]$. Let the scheduler activate (v, u) and (v, w) sequentially. This will cause $F_u[color_v]$ and $F_w[color_v]$ to flip once and $F_v[color_u]$ to flip twice, leaving it unchanged. Afterwards, (v, w) is synchronized but (u, v) is unsynchronized. Now continue with case 1.

Each subphase $\sigma(u)$ corrects the color of u and does not introduce any new color violations, so the configuration at the end of phase 1 satisfies the first safety condition.

In the second phase, we remove violations of the synchronization condition. For each unsynchronized edge (u, v), let the scheduler activate (u, v). If line 2 of the protocol is taken, let node u choose not to change its color. Because u will set $F_u[color_v] \leftarrow F_v[color_u]$, the edge (u, v) becomes synchronized. Since the configuration after phase 1 has no color violations, the neighbors of each node now all have distinct colors. Therefore synchronizing an edge does not introduce new unsynchronized edges. Because the second phase does not modify the colors, after all unsynchronized edges are corrected, the system is in a legitimate configuration. □

The following theorem, which is an immediate consequence of Lemmas 6.2 and 6.3 and Theorem 2.2, establishes the correctness of the protocol.

THEOREM 6.4. *For each d, there exists a constant-space self-stabilizing implementation of the 2-hop coloring behavior in undirected communication graphs of degree bounded by d.*

According to Corollary 3.6, there exists a deterministic version of this protocol. However, the full machinery of Corollary 3.6 is not needed to make the protocol work. In the following section, we show how a simpler deterministic protocol suffices.

6.2 Deterministic protocol

One way to turn the nondeterministic Protocol 2 into a deterministic one is to change the coloring rule at line 2, as shown in Protocol 3. Here, r_u is a local bit that flips whenever u acts as the initiator in an interaction. Consecutive repeated executions of line 2 cause u's color to cycle through all possible values, but an interaction when $r_u = 0$ will synchronize the interacting nodes without changing their colors, needed for the construction of Lemma 6.6.

LEMMA 6.5. *The trace of any execution of Protocol 3 starting from a legitimate configuration is in $2HC$.*

PROOF. Because the set of possible traces of Protocol 3 is a subset of that of Protocol 2, Lemma 6.5 follows from Lemma 6.2. □

LEMMA 6.6. *Starting from an arbitrary configuration, there exists a finite execution fragment of Protocol 3 that reaches a legitimate configuration.*

PROOF. We use a two-phase procedure similar to the proof of Lemma 6.3 to construct a path to a legitimate configuration from an arbitrary configuration.

As before, phase 1 consists of a sequence of subphases $\sigma(u)$, one for each node u. Each subphase consists of a sequence of interactions as described below. The

Protocol 3 Deterministic 2-hop coloring with degree bound d

For each node u:

State variables: $color_u$, F_u, and r_u.
Output: $color_u$

The interaction between an initiator u and a responder v:

```
1: if $F_u[color_v] \neq F_v[color_u]$ then          ▷ possibly conflicting colors
2:     $color_u \leftarrow (color_u + r_u) \bmod g$
3:     $F_u[color_v] \leftarrow F_v[color_u]$
4: else                                                ▷ valid coloring
5:     $F_u[color_v] \leftarrow \overline{F_u[color_v]}$
6:     $F_v[color_u] \leftarrow \overline{F_v[color_u]}$
7: end if
8: $r_u \leftarrow 1 - r_u$
```

subphases are constructed sequentially and refer to the configuration that results from the previous subphases.

Subphase $\sigma(u)$ is empty if u is non-violating at the start of the subphase, so assume now that u is violating. We construct a sequence of activations that change u's color so that it is no longer violating but do not change the color of any other node. $\sigma(u)$ consists of a sequence of segments $\tau_1(u), \tau_2(u), \ldots$. Each segment advances u's color to $(color_u + 1) \bmod g$ without changing the color of any other node. The subphase ends when u's color becomes distinct from that of all of its 2-hop neighbors.

Since u is violating, there are edges (u, v) and (v, w) such that $color_u = color_w$. Segment $\tau_i(u)$ consists of the activations described by the cases below, which are based on the values of $F_u[color_v]$, $F_v[color_u] = F_v[color_w]$, and $F_w[color_v]$:

(1) $F_u[color_v] \neq F_v[color_u]$. Let the scheduler activate (u, v).
(2) $F_u[color_v] = F_v[color_u]$ and $F_v[color_w] = F_w[color_v]$. Let the scheduler activate (v, w) followed by (u, v).
(3) $F_u[color_v] = F_v[color_u]$ and $F_v[color_w] \neq F_w[color_v]$. Let the scheduler activate (v, u), (v, w), and (u, v) sequentially.

In each case, some number of edges are activated, the last being (u, v). Only the last of those activations causes line 2 to be executed, so the only node whose color can possibly change is u. Moreover, u is the initiator of exactly one activation, so the sequence of activations causes bit r_u to flip.

If $r_u = 1$ at the time of the activation of (u, v), then $color_u$ is advanced and the segment is complete. If not, we repeat the above once. This time, r_u will be 1 when (u, v) is activated and $color_u$ will advance as desired. This completes the construction of segment $\tau_i(u)$, subphase $\sigma(u)$, and phase 1.

In the second phase, we remove violations of the synchronization condition. For each unsynchronized edge (u, v), there are two cases:

(1) $r_u = 0$: The scheduler activates (u, v). No colors change but (u, v) becomes synchronized. This step does not cause any other edges to lose synchronization

because the only state change is to $F_u[color_v]$ and v is non-violating; hence no other neighbor of u has the same color as v.

(2) $r_u = 1$: We need to change r_u to 0, so that the edge can be corrected as in case 1. There are two sub-cases:

(a) There exists some other synchronized edge (u, w) from u. Let the scheduler activate (u, w). This will set r_u to 0 without changing $color_u$. This step does not cause any other edges to lose synchronization because the only state changes are to $F_u[color_w]$ and $F_w[color_u]$, and both u and w are non-violating. We continue with case 1.

(b) No edge (u, w) is synchronized. Let the scheduler activate (u, v). This will change $color_u$. We now apply the activation sequence $\sigma(u)$ that results from applying the phase 1 construction to the current configuration. This will correct the color violation for node u without changing the color of any other node. It also synchronizes edge (u, v), so we are done.

We must show that no edge loses synchronization as a result of $\sigma(u)$. The fact that we are in case 2b means that no edge (u, w) is synchronized. Hence, no such edge can possibly lose synchronization as a result of $\sigma(u)$. No edges not involving u lose synchronization. We go back to the three possible edge activations in a segment, (v, u), (v, w), and (u, v) to see how they affect the F-values for non-u nodes.

These edge activations can flip $F_v[color_w]$ and $F_w[color_v]$. Since there were no color violations at the beginning of phase 2, no other neighbor of v has the same color as w (except for u), so changing $F_v[color_w]$ can not affect the synchronization status of v with any node other w (and u). Similarly, no other neighbor of w has the same color as v, so changing $F_w[color_v]$ can not affect the synchronization status of w with any node other than v.

It can be verified by inspection that the edge sequences specified by cases 1, 2, and 3 all leave $F_v[color_w]$ unchanged, and if they do change $F_w[color_v]$, it ends up the same as $F_v[color_w]$, thereby synchronizing v and w even if they were not synchronized beforehand. In no case does an edge lose synchronization.

Thus, after node u has been recolored, the first safety property again holds, no edge loses synchronization, and there exists a neighbor w of u that is synchronized with u. This concludes subcase 2b.

After all violating edges are corrected, the system is in a legitimate configuration. □

The correctness of Protocol 3 follows from Lemma 6.5, Lemma 6.6, and Theorem 2.2.

7. RING ORIENTATION

Given a graph colored by Protocol 2, we give a protocol that gives a sense of orientation to each node on an undirected ring. The orientation is globally consistent. In other words, each node has exactly one predecessor and exactly one successor; the predecessor and successor of each node are distinct; for any two nodes u and v, u is the predecessor of v if and only if v is the successor of u; for any edge (u, v), either u is the predecessor of v or v is the predecessor of u.

Definition 7.1. The *ring orientation behavior* RO on $G(V,E)$ is defined as the set of all constant traces $t = \lambda, \lambda, \ldots$ over an alphabet $C \times C \times C$. For all $v \in V$, $\lambda(v)$, in the form of $(c_v, c_{v,0}, c_{v,1})$, satisfies the following conditions:

(1) For all $v \in V$, $c_{v,0} \neq c_{v,1}$.
(2) For all $(u,v) \in E$, there exists $b \in \{0,1\}$ such that $c_v = c_{u,b} \wedge c_u = c_{v,\bar{b}}$.

In the above definition, we can think of c_u as the color of node u, $c_{u,0}$ as the color of its predecessor and $c_{u,1}$ as the color of its successor.

In the protocol, each node u has the following components:

$color_u$	the color of node u (we assume this value is provided by the input behavior $2HC$.)
$color_{u,0}$	the color of the predecessor
$color_{u,1}$	the color of the successor

Node u outputs $(color_u, color_{u,0}, color_{u,1})$.

Protocol 4 Orienting an undirected ring

For each node u:

Input: $color_u$
State variables: $color_u$, $color_{u,0}$, and $color_{u,1}$.
Output: $(color_u, color_{u,0}, color_{u,1})$

The interaction between an initiator u and a responder v:

```
1: if $color_v = color_{u,0}$ and $color_v \neq color_{u,1}$ then
2:     $color_{v,1} \leftarrow color_u$
3: else if $color_v = color_{u,1}$ and $color_v \neq color_{u.0}$ then
4:     $color_{v,0} \leftarrow color_u$
5: else
6:     $color_{u,0} \leftarrow color_v$
7:     $color_{v,1} \leftarrow color_u$
8: end if
```

A configuration is legitimate if its output assignment satisfies the requirement of RO.

Lemma 7.2. *In the executions of Protocol 4, given input behavior $2HC$, all reachable configurations from any legitimate configuration are also legitimate configurations.*

Proof. Let C be a legitimate configuration, that is, for all $(u,v) \in E$, there exists $b \in \{0,1\}$ such that $color_v = color_{u,b}$ and $color_u = color_{v,\bar{b}}$, and for all u, $color_{u,0} \neq color_{u,1}$. Depending on the value of b, the condition in either line 1 or line 3 is true, and the assignments in line 2 and 4 do not modify the states, since the components already have the assigned values. Therefore, starting from a legitimate configuration, the state of each node does not change. $\square$

LEMMA 7.3. *Starting from an arbitrary configuration given input behavior $2HC$, there exists a finite execution fragment of Protocol 4 that ends at a legitimate configuration.*

PROOF. Starting from an arbitrary configuration, consider an arbitrary node as the starting point and label the nodes sequentially as $0, 1, \ldots, n-1$. Let the scheduler activate edge $(0, 1)$. There are two possibilities for node 0 and 1 after the interaction:

(1) $color_{0,0} = color_1$ and $color_{1,1} = color_0$.

(2) $color_{0,1} = color_1$ and $color_{1,0} = color_0$.

We can assume 1 is true without loss of generality, because if 2 is true, we can just reverse the label of 0 and 1 and label the ring in the reverse direction. Let the scheduler activate $(1, 2)$, $(2, 3)$, $\ldots$, $(n-1, 0)$ sequentially. It must hold after each activation $(u, (u+1))$ that $color_{u,0} = color_{(u+1) \bmod n}$ and $color_{((u+1) \bmod n),1} = color_u$, because $2HC$ guarantees that $color_{u-1} \neq color_{u+1}$, and by induction the previous activation $(u-1), u$ guarantees that the condition at line 3 cannot be true.

After the above procedure, a legitimate configuration is reached. □

THEOREM 7.4. *Given the 2-hop-coloring input behavior, Protocol 4 is a constant-space self-stabilizing implementation of ring orientation.*

The correctness of Theorem 7.4 follows from Lemma 7.2 and Lemma 7.3.

8. SELF-STABILIZING SPANNING TREES IN REGULAR GRAPHS

Assuming the existence of a leader and the local addresses assigned by the 2-hop coloring protocol, a spanning tree rooted at the leader can be constructed in a self-stabilizing fashion in a regular graph of degree d. In this section we present a protocol that uses $O(\log D)$ bits of memory, where D is the diameter of the graph.

Definition 8.1. Let N be a set of labels and $\phi \notin N$ be a special symbol. The *spanning tree behavior ST* on graph $G(V, E)$ consists of all constant traces $t = \lambda, \lambda, \ldots$ such that:

(1) For $u \in V$, $\lambda(u)$ is a pair (c, p) where $c \in N$ and $p \in N \cup \{\phi\}$, and there exists a unique $r \in V$ such that the second component of $\lambda(r)$ is ϕ.

(2) The projection of λ on the first component of the outputs $\Pi_c(\lambda)$ satisfies the 2-hop coloring behavior defined in Section 6.

(3) For all $u_0 \neq r$, there exists a path $u_0, u_1, \ldots, v_k$, where $u_i \in V$, $u_k = r$, $\lambda(u_i) = (c_i, p_i)$ and $p_i = c_{i+1}$ for all $0 \leq i < k$.

Informally, N is the set of possible colors of nodes. If $\lambda(u) = (c, p)$, c is the color of u and p is the color of its parent in the spanning tree. The network is 2-hop colored. For the root node r in the spanning tree, $p = \phi$.

We define the *first spanning tree* of a 2-hop-colored graph with a unique leader to be the spanning tree satisfying the following conditions:

(1) The leader is the root of the tree.

(2) The parent of each node is the neighbor closest to the root. Ties are broken by choosing the neighbor with smallest color.

It is easy to see that if the coloring and the leader are fixed, the first spanning tree is unique. Protocol 5 is a self-stabilizing protocol which constructs the first spanning tree in any properly 2-hop-colored regular graph given a root.

First of all, each node needs to know its neighbors. We let each node u keep a queue $neighbors_u$ of size d. When u interacts with another node v, it checks if the color of v ($color_v$) is in its queue. If not, it pushes $color_v$ into the queue. If the queue has more than d elements, the oldest value is removed. u will eventually have the d distinct colors of all its neighbors in $neighbors_u$.

To find the parent in the first spanning tree as defined above, node u tracks the smallest known distance to the root in a variable $dist_u$ and tracks the color of its current parent in a variable $parent_u$. When u interacts with another node v, if u is the root, u records ϕ as its current parent and sets $dist_u$ to 0. Otherwise u updates its state to record v as its new parent if any of the following is true:

(1) $parent_u$ is not in $neighbors_u$, which means either or both of $parent_u$ and $neighbors_u$ are incorrect.

(2) $dist_u > dist_v + 1$, which means the path through v to the root is the shortest known to u.

(3) $dist_u = dist_v + 1$ and $parent_u > color_v$, which means v does not give a shorter path to the root, but the color of v of is smaller than the color of the current parent of u given a fixed ordering of colors.

If at the end of the interaction v is the current parent of u, Node u sets $dist_u$ to $dist_v + 1$.

Protocol 5 is a formal specification. The variables of each node u have the following meanings:

$neighbors_u$	a queue containing the colors of the d most-recently-seen neighbors.
$dist_u$	a nonnegative integer, to store length of the known shortest path from the root.
$parent_u$	color of the parent node.
$color_u$	the color of node u, provided by the input behavior $2HC$.
$status_u$	Either L (leader) or N (nonleader), provided by the input behavior LE.

The operations on a queue are defined below:

enqueue(item)	inserts the item to the front of the queue.
dequeue()	removes the last item from the back of the queue.
$\in$ or $\notin$	tests occurrence of an element in the queue.

Theorem 8.2. *Given the input behaviors LE and $2HC$, Protocol 5 is a self-stabilizing implementation of output behavior ST for all regular graphs of degree d.*

Protocol 5 Self-stabilizing spanning tree in a regular graph

For each node u:

Inputs: $color_u$ and $status_u$

State variables: $dist_u$ and $parent_u$

Output: $(color_u, parent_u)$

The interaction between an initiator u and a responder v, when (u, v) is activated:

```
1: if $color_v \notin neighbors_u$ then
2:     $neighbors_u$.enqueue($color_v$)
3: end if
4: while $|neighbors_u| > d$ do                ▷ keep $d$ recently-seen neighbors
5:     $neighbors_u$.dequeue()
6: end while
7: if $status_u = L$ then
8:     $dist_u \leftarrow 0$;
9:     $parent_u \leftarrow \phi$
10: else if $(parent_u \notin neighbors_u)$
11:         $\vee(dist_u > dist_v + 1)$
12:         $\vee(dist_u = dist_v + 1 \wedge parent_u > color_v)$ then
13:     $parent_u \leftarrow color_v$
14: end if
15: if $parent_u = color_v$ then
16:     $dist_u \leftarrow dist_v + 1$
17: end if
```

Proof. Given input behaviors $2HC$ and LE, we may assume that the interaction graph G is properly 2-hop colored with one node marked L and all other nodes marked N, and we use "leader" and "root" interchangeably afterwards. Let T denote the unique first spanning tree of G.

For Protocol 5, a configuration is legitimate if the following conditions hold:

(1) For any node u, $neighbors_u$ stores the d distinct colors of its neighbors.

(2) For any non-root node u, $parent_u$ stores the color of its parent in T, and $dist_u$ is equal to $dist_u^T$, the length of the path from u to the root in T.

(3) For the root node r, $parent_r = \phi$ and $dist_r = 0$.

From the definition of T, it is obvious that no interaction can cause a state change in a legitimate configuration.

Now we show that a legitimate configuration is reachable from an arbitrary configuration. First, activate every edge in each direction once. The queue of neighbors of each node now stores the d colors of its neighbors. The root is in the correct state with its parent set to ϕ, the distance to the root set to 0, and for each non-root node u, $parent_u$ stores the color of one of its neighbors. Define graph H to consist of the edges (u, v) such that $parent_u = color_v$.

Among all edges $(u, v) \in H - T$, choose one that minimizes $dist_u^T$. Let w be u's parent in T. Thus, $(u, v) \in H$ and $(u, w) \in T$. Inductively, we assume that all nodes u' with $dist^T_{u'} < dist^T_u$ have $parent_{u'}$ set correctly, that is, $parent_{u'} = color_{v'}$ where $(u', v') \in T$.

We activate a series of edges that changes u's parent to w but does not change the parent of any other node. There are two cases:

(1) In graph H, the root is reachable from u. First, activate the edges on the path from the root to w in H sequentially. These edges are in both T and H since all nodes u' on that path have $dist^T_{u'} < dist^T u$. Because each node will set its distance variable to be the distance variable of its parent plus one, $dist_w$ will be the real distance $dist^T_w$. Next, activate the edges on the path from the root to u in H sequentially. After that it must hold that $dist_u$ is the length of the path from root to u in H. Since T is the first spanning tree, we know that either $dist_u > dist_w + 1$ is true, or $dist_u = dist_w + 1$ and $color_v > color_w$ are true. In either case, activate (u, w), and u will mark w as the new parent.

(2) In graph H, the root is not reachable from u. Let's only look at H, and let the component C consist of u and the nodes reachable from it. C does not contain the root. Because all nodes in C have out-degree one, there must be a directed cycle in C. By the definition of H, every node in the cycle marks the next node as its parent. By activating the edges in the cycle, the $dist$ values of the nodes in the cycle can be increased to arbitrarily large values, because for any edge $(u, v) \in H$, u sets $dist_u = dist_v + 1$ when (u, v) is activated. By activating the edges on the path from any node in the cycle to u, the large $dist$ value will be propagated back to u. When $dist_u > dist_w + 1$ activate (u, w), and u will mark w as the new parent.

This process corrects the $parent_u$ mark for u. It does not change the parent mark for any other node because activating an edge $(u, x) \in H$ can never change $parent_u$. The process is repeated until $H = T$.

Finally, activate the edges in T in the order of a breadth-first traversal to make sure that every node u updates $dist_u$ to the correct value. The configuration is now legitimate. □

We remark that a traversal of the tree can simulate an oriented ring. Therefore, token-circulation can be done in a regular graph with a distinguished leader by composing the ring token-circulation protocol with the 2-hop coloring protocol and the spanning tree protocol.

9. LEADER ELECTION

Two of the above protocols assume a pre-designated special node. In our model, self-stabilizing leader election is possible in some classes of interaction graphs and impossible in others. In this section, we first describe a family of leader-election protocols in oriented rings. We also present an impossibility result for leader election in general graphs. The formal definition of the leader-election behavior (LE) is given in Section 5.

We first consider rings of odd size. Supposing each node has a *label* bit, we call a maximal sequence of alternating labels a *segment*. Since the size of the ring is odd, there is at least one pair of adjacent nodes with the same label. We define the *head* and *tail* of a segment in the natural way according to the orientation of the ring. For example, the ring $0 \to 1 \to 0 \to 0 \to 1 \to 1 \to 0 \to 1 \to 0$ has 3 segments: $0 \to 1 \to 0$, $0 \to 1$, and $1 \to 0 \to 1 \to 0$. One edge of the form $(0,0)$ or $(1,1)$ connects the tail of one segment to the head of another segment. We call such edges *barriers*.

The protocol consists of several parts. At the base is the "unstable clock" protocol, in which the barriers move forward (clockwise) around the ring . Each barrier advances by flipping the label bit of the second node on the barrier (the head of the next segment). It is easy to verify that when two barriers collide, they cancel out each other (both of the barriers disappear). Because the ring size is odd, there is always at least one barrier, and there exists a sequence of activations that remove all but one barrier. By fairness, eventually there is a single barrier which rotates clockwise around the ring forever.

The remainder of the protocol manipulates the leader marks and two kinds of tokens, *bullet* and *probe*. Probes move faster than barriers. (we do not allow a barrier to go past a probe from behind). Probes are sent out by the barrier in a clockwise direction and absorbed by any leader they run into. If a probe makes it all the way back to the barrier, it is converted to leader (just behind the barrier, which we can imagine is between the two endpoints of the barrier edge). Leaders fire bullets counterclockwise around the ring. Bullets are absorbed by the barrier, but they kill any leaders they encounter along the way (that is, they remove the leader mark).

Call a configuration "clean" if it contains exactly one barrier, exactly one leader, and there are no *bullet* or *probe* marks on any node in the interval starting from the leader and proceeding clockwise to the barrier. Thus, any *bullet* and *probe* marks are confined to the interval starting from the barrier and proceeding clockwise to the leader. As the barrier rotates, this region gets squeezed smaller and smaller until finally the barrier passes leader, at which point there are no *bullet* or *probe* marks at all. Protocol 6 is the detailed specification. Each node u outputs L if $\text{leader}_u = 1$, otherwise it outputs N.

The state variables of each node u have the following meanings:

leader_u	1 if u is a leader, 0 otherwise.
$label_u$	the label of u.
$probe_u$	1 if u holds a probe token, 0 otherwise.
$phase_u$	alternates between 0 and 1 to make each barrier alternate between firing a probe and moving forward.

LEMMA 9.1. *Assume Protocol 6 runs in a ring of odd size. Then all configurations reachable from a clean configuration are also clean, and the same node is marked* leader *in each.*

PROOF. No *probe* ever encounters the barrier, (from behind), because there are no *probe* marks anywhere in the region between leader and the barrier, hence, no new leader is created. No *bullet* ever encounters the leader because there are no *bullet* marks anywhere in the region starting from the barrier and going counterclockwise (the direction of the bullet) to the barrier, hence the leader is never

Protocol 6 Leader election in a ring of odd size

For each node u:

State variables: $bullet_u$, leader_u, $label_u$, $probe_u$, and $phase_u$.
Output: If $leader_u = 1$, output L; Otherwise, output N.

The $phase_u$ variable is used to make each barrier alternate between generating a probe and moving itself forward. The following specifies the interaction between an initiator u and a responder v, when (u, v) is activated:

```
if $label_u = label_v$ then                  ▷ This is a barrier edge.
    if $probe_u = 1$ then                    ▷ A probe hits a barrier
        $\text{leader}_u \leftarrow 1$       ▷ generate a leader
        $probe_u \leftarrow 0$
    end if
    if $phase_u = 0$ then                    ▷ generate a probe
        $phase_u \leftarrow 1$
        if $\text{leader}_v = 0$ then
            $probe_v \leftarrow 1$
        end if
    else if $probe_v = 0$ then               ▷ advance the barrier
        $label_v \leftarrow \overline{label_v}$
        $phase_v \leftarrow 0$
        $bullet_v \leftarrow 0$              ▷ absorb incoming bullets
    end if
else if $\text{leader}_v = 1$ then           ▷ non-barrier edge, $v$ is leader
    if $bullet_v = 1$ then                   ▷ If there is a bullet at $v$
        $\text{leader}_v \leftarrow 0$       ▷ $v$ is killed
    else
        $bullet_u \leftarrow 1$              ▷ otherwise $v$ fires a bullet to $u$
        $probe_u \leftarrow 0$               ▷ and absorb a probe if any.
    end if
else                                         ▷ $v$ is not a leader.
    if $bullet_v = 1$ then                   ▷ advance a bullet
        $bullet_v \leftarrow 0$
        $bullet_u \leftarrow 1$
    end if
    if $probe_u = 1$ then                    ▷ advance a probe
        $probe_u \leftarrow 0$
        $probe_v \leftarrow 1$
    end if
end if
```

killed. Newly created *bullet* and *probe* marks are both confined to the region from the barrier to the leader. □

LEMMA 9.2. *Assume Protocol 6 runs in a ring of odd size. For any configuration C there exists a clean configuration C' reachable from C.*

Proof. First, pick a barrier edge and advance the barrier until it collides with another barrier, and both barriers are removed. Repeat until there is only one barrier left. This must occur since every ring of odd size always has at least one barrier. Next, take any bullet in the forbidden region starting from the barrier and proceeding counterclockwise to the first leader (or the entire ring if no node is marked leader) and propagate the bullets counterclockwise around the ring until they are absorbed by the barrier. Some or all leaders may die in the process. If any leader remains, take the farthest leader from the barrier (in the counterclockwise direction), fire a bullet, and propagate it until it is absorbed by the barrier. Now at most one leader remains. Next, let the barrier create a *probe* mark, then propagate all *probe* marks clockwise around the ring until they are absorbed by the leader or they encounter the barrier and are converted to leader. At this point, we have a ring with one barrier, one leader, and no other marks, so it is clean. □

Theorem 9.3. *Protocol 6 is a constant-space self-stabilizing implementation of the leader-election behavior on all rings with odd sizes.*

Proof. From Lemma 9.2, it follows from our fairness condition that every fair computation contains a clean configuration. From Lemmas 9.1, all configurations following the first clean configuration are also clean and have the same node marked as leader. □

This protocol is a special case of a family of protocols. For any ring of size n, we can pick an integer $k > 1$ that is relatively prime to n. Each node is labeled by an integer between 0 and $k - 1$ inclusive. Call an edge (u, v) a "barrier" if $label_u + 1 \not\equiv label_v \pmod{k}$. Because k is relatively prime to n, there is always at least one barrier. To advance a barrier edge (u, v), set $label_v \leftarrow label_u + 1 \bmod k$. Call this protocol P_k, then the protocol we detailed in this section is P_2. Thus we have a family of protocols P_2, P_3, ... such that for any ring, P_k accomplishes self-stabilizing leader election whenever k does not divide the size of the ring.

Theorem 9.4. *For each integer $k \geq 2$, there is a constant-space self-stabilizing implementation of the leader-election behavior on all rings whose sizes are not multiples of k.*

Finally, we present the following impossibility result:

Theorem 9.5. *There does not exist a self-stabilizing protocol for leader election in interaction graphs with general topology.*

Proof. Assuming such a protocol A exists, we consider how it would behave in directed lines. Let e be an arbitrary edge. If e were removed, the interaction graph would become two directed lines, and by the correctness assumption of A, the two shorter lines would each elect a leader. Therefore from any configuration C there is a reachable configuration C' in which there are two leaders, because from C the scheduler just stops activating e and only activates other edges for a certain amount of time to reach C'. By fairness, in any fair execution of A, some configuration C' with two leaders occurs infinitely often. Therefore the output trace of any fair execution of A cannot have a suffix in the behavior LE. □

A class C of graphs is *simple* if there does not exist a graph in C which contains two disjoint subgraphs that are also in C. Notable simple classes of graphs include rings, or, more generally, connected degree-d regular graphs. Directed lines, connected graphs with a certain degree bound and strongly connected graphs are non-simple classes of graphs. The proof above shows that there is no self-stabilizing leader election protocol that works for all the graphs in any non-simple class.

10. CONCLUSION AND OPEN PROBLEMS

In this paper, we extended the population protocol model of [Angluin et al. 2006] to allow for inputs at each step, and we defined general classes of behaviors. We studied self-stabilization protocols for token-circulation, 2-hop coloring, ring orientation, spanning tree, and leader election in this extended model.

We remark that one of the applications of the self-stabilizing protocols is to combine them with the protocols in [Angluin et al. 2006; Angluin et al. 2005] to compute algebraic predicates or graph properties, with the additional benefit of transient-fault tolerance. For instance the token-circulation protocol could be augmented to compute predicates such as $n > k$ or expressions like $n \bmod k$ in regular graphs in which n is the size of the network and k is a constant.

The leader election protocol we presented in this paper depends on the size of the ring. There are impossibility results and space bounds on self-stabilizing leader election in general rings in various other models [Dijkstra 1974; Beauquier et al. 1999]. Because of the difference between our model and those of the previous papers, those results cannot be easily extended to our model. The existence of a uniform constant-space leader election protocol on the class of all rings or on the class of communication graphs that are regular of degree $d > 2$ is still open for future research.

REFERENCES

Angluin, D., Aspnes, J., Chan, M., Fischer, M. J., Jiang, H., and Peralta, R. 2005. Stably computable properties of network graphs. In *Distributed Computing in Sensor Systems: First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USA, June/July, 2005, Proceedings*, V. K. Prasanna, S. Iyengar, P. Spirakis, and M. Welsh, Eds. Lecture Notes in Computer Science, vol. 3560. Springer-Verlag, Berlin / Heidelberg, 63–74.

Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M. J., and Peralta, R. 2006. Computation in networks of passively mobile finite-state sensors. *Distributed Computing 18,* 4, 235–253. DOI 10.1007/s00446-005-0138-3.

Angluin, D., Aspnes, J., Eisenstat, D., and Ruppert, E. 2007. The computational power of population protocols. *Distributed Computing 20,* 4 (Nov.), 279–304.

Attiya, H., Bar-Noy, A., Dolev, D., Koller, D., Peleg, D., and Reischuk, R. 1987. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 337–346.

Beauquier, J., Gradinariu, M., and Johnen, C. 1999. Memory space requirements for self-stabilizing leader election protocols. In *Eighteenth ACM Symposium on Principles of Distributed Computing.* Association for Computing Machinery, New York, NY, USA, 199–207.

Dijkstra, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Communications of the ACM 17,* 11, 643–644.

Dolev, S., Israeli, A., and Moran, S. 1997. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems 8*, 424–440.

Gouda, M. G. and Haddix, F. F. 1996. The stabilizing token ring in three bits. *J. Parallel Distrib. Comput. 35,* 1, 43–48.

Gradinariu, M. and Johnen, C. 2001. Self-stabilizing neighborhood unique naming under unfair scheduler. *Lecture Notes in Computer Science 2150*, 458–465.

Griggs, J. R. and Yeh, R. K. 1992. Labelling graphs with a condition at distance 2. *SIAM Journal on Discrete Mathematics 5,* 4, 586–595.

Herman, T. 1990. Probabilistic self-stabilization. *Information Processing Letters 35,* 2, 63–67.

Herman, T. and Tixeuil, S. 2004. A distributed TDMA slot assignment algorithm for wireless sensor networks. *Lecture Notes in Computer Science 3121*, 45–58.

Higham, L. and Myers, S. 1999. Self-stabilizing token circulation on anonymous message passing rings. Tech. rep., University of Calgary.

Itkis, G. and Levin, L. A. 1994. Fast and lean self-stabilizing asynchronous protocols. In *Proceeding of 35th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 226–239.

Itkis, G., Lin, C., and Simon, J. 1995. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*. Springer-Verlag, London, UK, 288–302.

Johnen, C. 2004. Bounded service time and memory space optimal self-stabilizing token circulation protocol on unidirectional rings. In *Procedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 52a.

Mayer, A., Ofek, Y., Ostrovsky, R., and Yung, M. 1992. Self-stabilizing symmetry breaking in constant-space (extended abstract). In *Proc. 24th ACM Symp. on Theory of Computing*. Association for Computing Machinery, New York, NY, USA, 667–678.

Moscibroda, T. and Wattenhofer, R. 2005. Coloring unstructured radio networks. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM Press, New York, NY, USA, 39–48.

Received March 5, 2007