Yale University
Department of Computer Science

P.O. Box 208205
New Haven, CT 06520–8285

Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents[1]

Michael Fischer Hong Jiang[2]

YALEU/DCS/TR-1370
October 2006

[1] This is a slightly-revised preprint of a paper to appear in A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305.

[2] Supported by NSF grant ITR-0331548

Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents*

Michael Fischer and Hong Jiang**

Department of Computer Science, Yale University

Abstract. This paper considers the self-stabilizing leader-election problem in a model of interacting anonymous finite-state agents. Leader election is a fundamental problem in distributed systems; many distributed problems are easily solved with the help of a central coordinator. Self-stabilizing algorithms do not require initialization in order to operate correctly and can recover from transient faults that obliterate all state information in the system. Anonymous finite-state agents model systems of identical simple computational nodes such as sensor networks and biological computers. Self-stabilizing leader election is easily shown to be impossible in such systems without additional structure.
An *eventual leader detector* $\Omega?$ is an oracle that eventually detects the presence or absence of a leader. With the help of $\Omega?$, uniform self-stabilizing leader election algorithms are presented for two natural classes of network graphs: complete graphs and rings. The first algorithm works under either a local or global fairness condition, whereas the second requires global fairness. With only local fairness, uniform self-stabilizing leader election in rings is impossible, even with the help of $\Omega?$.

Keywords: anonymous, failure detector, fairness, finite-state, impossibility result, leader election, population protocols, ring network, self-stabilization, sensor networks.

1 Introduction

Leader election is a fundamental problem in distributed systems. Many problems that are hard otherwise become easy to solve once a central coordinator is available. In reality, the availability and reliability of a leader both depend on a variety of factors: the feasibility to deploy or elect a leader, the possibility that an existing leader crashes, and the possibility that transient faults generate multiple leaders.

In many scenarios, a reasonable expectation is that when the network eventually becomes well-behaved and remains so, a leader is elected and remains reliable. This behavior is captured by failure detector Ω [1] also known as an

* This is a slightly-revised preprint of a paper to appear in A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305.

** Supported by NSF grant ITR-0331548

eventual leader elector. With Ω, every process i has a local oracle $\texttt{leader}_i$. When invoked, $\texttt{leader}_i$ returns a process ID which process i considers to be its current leader. Ω guarantees that there is a time after which all processes have the same non-faulty leader. Ω is important because it was shown to be the weakest failure detector required to solve consensus in the conventional model of asynchronous distributed systems [1], and it has been used in many other algorithms.

Ω presupposes a model in which agents have unique process ID's. In this paper, we study a model of distributed systems called *population protocols* which was developed in a series of papers [2–4]. A network consists of an unbounded but finite population of identical anonymous finite-state agents. The protocols we present are uniform over natural classes of networks: They are independent of the network size, and the agents do not need to know the size of the network. The agents in our model are strongly anonymous. Not only do agents lack unique process ID's, but an agent cannot even determine whether two distinct messages are from the same process, nor can it direct two outgoing messages to the same recipient. By way of contrast, some related work on anonymous networks assumes an underlying port-to-port communication model in which processes are permanently assigned to ports, and a process sends and receives messages through distinguished ports. Such a model gives agents the ability to tell whether a set of messages come from different neighbors and to direct messages to the same or distinct neighbors, which is generally impossible in our model. Identical devices are easy to manufacture in large quantity. In addition, population protocols use $O(1)$ space per node, which is highly desirable in networks of memory-limited devices such as ad hoc mobile networks.

We introduce $\Omega?$, an analog of Ω appropriate to anonymous networks, which we call an *eventual leader detector*. Instead of electing a leader, $\Omega?$ simply reports to each agent a guess about whether or not one or more leaders are present in the network. The guess may be correct or not, and different inconsistent guesses may be reported to different agents. The only guarantee is that from some point onward in any infinite execution, if there is continuously a leader, or if there is continuously no leader, $\Omega?$ eventually accurately reports that fact to each agent.

Using $\Omega?$, we give uniform self-stabilizing leader election algorithms for fully connected networks (assuming local fairness) and for rings (assuming global fairness). We also show that uniform leader election is impossible in rings with only local fairness, even with the help of $\Omega?$. The different fairness conditions are defined in section 3.1

2 Related work

A self-stabilizing algorithm does not depend on initialization of process states, and an execution of a self-stabilizing algorithm converges to a set of pre-defined stable configurations starting from any arbitrary configuration. The first self-stabilizing algorithms are introduced by Dijkstra[5]. Schneider[6] presents a survey on early research on self-stabilization.

Itkis, Lin, and Simon [7] 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. Higham and Myers [8] 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 [9] present a randomized 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 [10] 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. They also prove that a non-constant lower bound on space is required by a (deterministic or randomized) self-stabilizing leader-election protocol on unidirectional anonymous rings under an unfair daemon. Based on the observation that in a stabilized system, a transient fault usually affect a small number of processes, Ghosh and Gupta [11] introduce a self-stabilizing leader-election algorithm that recovers quickly from small scale transient faults. Their algorithm assumes the existence of unique IDs. Fernández, Jiménez, and Raynal [12] present two eventual leader-election algorithms in networks where nodes have limited global information. Their algorithms are implementations of Ω in a hybrid model and require unique and totally-ordered IDs. Angluin et al. [4] present a non-uniform leader-election algorithm for rings in the population protocols model. They also show in the same paper that there does not exist a self-stabilizing leader-election protocol for general connected networks.

Chandra and Toueg [13] introduce the concept of unreliable failure detectors and study how they can be used to solve the asynchronous consensus problem with crash failures. In a related paper, Chandra, Hadzilacos, and Toueg [1] prove that one of the failure detectors in [13] is the weakest failure detector for solving asynchronous consensus with a majority of reliable processes. They also show that Ω is a weakest failure detector with which one can solve consensus. Aguilera et al. present an algorithm to implement Ω and to solve consensus in partially synchronous systems [14]. Ω can be implemented in a system with up to f process crashes, if there exists some correct process with f outgoing links that are eventually timely. The focus of these papers is the consensus problem, so the underlying network is assumed to be fully connected.

3 Model and Definitions

We introduce the population-protocol model to the extent required to present the results in this paper. A more detailed description is available in [4].

We represent a network by a directed graph $G = (V, E)$ with n vertices numbered 0 through $n-1$ and no multi-edges or self-loops. Each vertex represents

a finite-state sensing device called an agent, 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*. An "undirected" network refers to a communication graph in which edge (v, u) is present if and only if edge (u, v) is present. The number associated with each node is used solely for the ease of description and is not known to the agents.

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*. A transition does not necessarily cause either of the nodes to change its state. 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. In this paper, an agent i interacts with its leader detector through the input port.

A *configuration* is a mapping $C : V \to Q$ specifying the state of each device in the network, and an *input assignment* is a mapping $\alpha : V \to X$. 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. The set Z is called the *alphabet* of T_G. If $Z = X$, then each λ_i is an input assignment, and we say T_G is an *input trace* of the protocol.

An *action* is a pair $\sigma = (r, e)$, where r is a transition $((p, x), (q, y)) \to (p', q')$ of δ and $e = (u, v)$ is an edge of G. Let C and C' be configurations, α be an input assignment, and u, v be distinct nodes. We say that σ is *enabled* in (C, α) if $C(u) = p$, $\alpha(u) = x$, $C(v) = q$, and $\alpha(v) = y$. We say that (C, α) *goes to* C' *via* σ, denoted $(C, \alpha) \xrightarrow{\sigma} C'$, if σ is enabled in (C, α), $C'(u) = p'$, $C'(v) = q'$, and $C'(w) = C(w)$ for all $w \in V - \{u, v\}$. In words, C' is the configuration that results from C by applying the transition rule r to the node pair e. Finally, we say that (C, α) *can go to* C' *in one step*, denoted $(C, \alpha) \to C'$, if $(C, \alpha) \xrightarrow{\sigma} C'$ for some action σ, and we say that σ is *taken* during that step. It is possible for more than one action to be taken during the same step.

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.

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, $(C_i, \alpha_i) \to C_{i+1}$. In the rest of this paper, all occurrences of "execution" refer to an infinite sequence as defined here. 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 $E = (C_0, \alpha_0), (C_1, \alpha_1), \ldots, (C_i, \alpha_i), \ldots$ be an execution of P. We define the *output trace* of an execution as $OT(E) = O(C_0), O(C_1), \ldots, O(C_i), \ldots$.

3.1 Fairness

We consider fairness conditions of different strengths. Let $E = (C_0, \alpha_0), (C_1, \alpha_1), \ldots, (C_i, \alpha_i), \ldots$ be an execution. The following conditions apply to E.

Strong global fairness For every C, α, and C' such that $(C, \alpha) \rightarrow C'$, if $(C, \alpha) = (C_i, \alpha_i)$ for infinitely many i, then $(C_i, \alpha_i) = (C, \alpha)$ and $C_{i+1} = C'$ for infinitely many i. (Hence, the step $(C, \alpha) \rightarrow C'$ is taken infinitely many times in E.)

Strong local fairness For every action σ, if σ is enabled in (C_i, α_i) for infinitely many i, then $(C_i, \alpha_i) \xrightarrow{\sigma} C_{i+1}$ for infinitely many i. (Hence, the action σ is taken infinitely many times in E.)

Global fairness asserts that each *step* $(C, \alpha) \rightarrow C'$ that can be taken infinitely often is actually taken infinitely often. By way of contrast, local fairness only asserts that each *action* σ that can be taken infinitely often is actually taken infinitely often. This differs from global fairness in the case of an action that is enabled infinitely often in more than one context. Global fairness would insist that it be taken infinitely often in all such contexts, whereas local fairness only requires that it occur infinitely often in one such context. For example, if σ is enabled in both (C_1, α_1) and (C_2, α_2), where $(C_1, \alpha_1) \neq (C_2, \alpha_2)$, an execution in which σ was never taken from (C_2, α_2) would not be globally fair, but it would be locally fair if σ were taken infinitely often from (C_1, α_1).

Theorem 1. *Global fairness implies local fairness.*

Proof. Suppose E satisfies strong global fairness. Because there are only finitely many distinct (C_i, α_i) pairs in E, if σ is enabled in (C_i, α_i) for infinitely many i, then σ is enabled in some particular (C, α) that occurs infinitely often in E. Let $(C, \alpha) \xrightarrow{\sigma} C'$. By strong global fairness, the step $(C, \alpha) \rightarrow C'$ is taken infinitely many times in E; hence, E satisfies strong local fairness.

These fairness definitions talk about certain steps that must be taken infinitely many times in E. For many purposes, it is immaterial whether a goal configuration C' is reached in one step or in many. This leads us to define corresponding weak fairness conditions.

Weak global fairness For every C, α, and C' such that $(C, \alpha) \rightarrow C'$, if (C, α) occurs infinitely often in E, then C' occurs infinitely often in E.

Weak local fairness For every action σ, if σ is enabled infinitely often in E, then there exist C, α, C' such that $(C, \alpha) \xrightarrow{\sigma} C'$, (C, α) occurs infinitely often in E, and C' occurs infinitely often in E.

The weak forms of fairness do not insist that particular steps occur infinitely often in E but only that the configurations that would result from those steps occur infinitely often. Thus, whereas strong fairness insists that a particular action occurs in a single step, weak fairness allows the configuration that would result from that action to be reached in many steps.

Obviously, the weak forms of fairness are implied by the corresponding strong forms. But the relationship between the weak and strong forms is even closer.

Theorem 2. *Every execution sequence that satisfies weak global (resp. local) fairness has an infinite subsequence that satisfies strong global (resp. local) fairness. Moreover, the sets of infinitely occurring pairs (C, α) are the same in both sequences.*

Proof (Sketch). The intuition is that if $(C_i, \alpha_i) \rightarrow C_j$ for $j > i$, then the segment $(C_{i+1}, \alpha_{i+1}), \ldots, (C_{j-1}, \alpha_{j-1})$ can be removed from E and the result is still an execution sequence. In the new sequence, (C_i, α) is adjacent to C_j, so $(C_i, \alpha_i) \rightarrow C_j$ occurs as a single step as required by strong fairness. Details are left to the full paper.

Discussion A fair question to ask is, "Which is the 'right' definition of fairness?" In light of Theorem 2, it makes little difference whether one works with the strong or weak forms of fairness, for a weakly fair execution has embedded in it a corresponding strongly fair execution. In subsequent sections, we do not explicitly distinguish between the strong and weak versions of the fairness conditions when the difference is immaterial.

Whether global or local fairness is more realistic depends on how scheduling decisions are made. If the next step to take is chosen randomly, with each possible step having a non-zero probability of being chosen, then a globally fair execution will result with probability 1.

However, systems are often viewed as consisting of a collection of semi-autonomous components. The scheduler activates each component infinitely often, but the scheduling decision is not assumed to be independent of the states of the other components. Thus, component A might be permitted to execute when component B is in state 1 but not when it is in state 2. As long as A is given infinitely many chances to run, the scheduler would be considered to be fair, even though A never gets to run at a time when B is in state 2. For such a system, local fairness (in one of its many varieties) is the appropriate notion of fairness.

3.2 Behavior, Implementation and Self-stabilization

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 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 a behavior $B(Z)$, we define the corresponding *stable behavior* $B^s(Z)$: $T \in B^s$ if and only if Z is T's alphabet, and there exists $T' \in B$ such that T' is a suffix of T. Thus, an execution in a stable behavior may have 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.

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 non-leader mark N in every configuration.

3.3 Eventual Leader Detector $\Omega?$

A failure detector is a kind of oracle that provides some information to the system that it is unable to compute on its own, thereby extending the power of the system. Traditionally, failure detectors have been viewed as diagnostic devices that test nodes and inform the system when failures are detected, hence the name. However, failure detectors are a more general concept. In this paper, we use them to supply global semantic information about the protocol, namely, whether or not a leader is present in the system. Our failure detectors are weak in the sense that they do not respond immediately to the presence or absence of a leader but only after some indeterminate delay. Moreover, they do not report their findings to all nodes simultaneously, so some nodes might learn of the loss or gain of a leader before others do. Traditionally, failure detectors are modeled as local procedure calls in each process. In this paper we model a failure detector as a black box. Instead of each node invoking a local procedure, the failure detector feeds inputs to each node at each step.

The *eventual leader detector* $\Omega?$ supplies a Boolean input to each process at each step so that the following conditions are satisfied by every execution E:

1. If all but finitely many configurations of E lack a leader, then each process receives input `false` at all but finitely many steps.
2. If all but finitely many configurations of E contain one or more leaders, then each process receives input `true` at all but finitely many steps.

3.4 Implementation of $\Omega?$

The weak guarantees of $\Omega?$ allow it to be simply implemented in practice using timeouts. Each leader periodically propagates a "keep-alive" signal. Each agent keeps a timer and resets the timer whenever it receives a signal from a leader. On timeout, the agent sets the leader detector flag to `false` to indicate the absence of a leader. It sets the flag back to `true` whenever it receives a signal from a leader. In a good environment where the links are reliable and timely, each agent will eventually detect the absence or presence of leaders correctly. In an adverse environment where nodes malfunction and links may drop or unduly delay messages, the leader detector may give incorrect answers and the system may become unstable. (For example, multiple leaders may be generated.) However, eventually after the environment becomes good again, the leader detector will produce correct information and the system will become stable.

4 Leader Election in Complete Network Graphs

We give a simple leader-election algorithm for complete network graphs using a leader detector $\Omega?$. Each node has a memory slot that can hold either a leader mark "♟" or nothing "$-$" for a total of two states. Each node receives its current input `true` (T) or `false` (F) from $\Omega?$. A non-leader becomes a leader, when the leader detector signals the absence of a leader, and the responder is not a leader. When two leaders interact, the responder becomes a non-leader. Otherwise, no state change occurs.

We formally describe the algorithm by pattern rules which are matched against the state and input of the initiator and responder, respectively. If the match succeeds, the states of the two interacting nodes are replaced by the respective states on the right side of the rule. In performing the match, "$*$" is a "don't care" symbol that always matches the slot or the input. On the right hand side, "$*$" specifies that the contents of the corresponding slot do not change. If no explicit rules match, a null transition in which neither node changes state is implied. Therefore every (configuration, input assignment) pair has an admissible successor.

Algorithm 1

$$\begin{array}{ll} \textit{Rule 1.} & ((♟, *), (♟, *)) \to ((♟), (-)) \\ \textit{Rule 2.} & ((-, \mathsf{F}), (-, *)) \to ((♟), (-)) \\ \textit{Rule 3.} & ((-, \mathsf{T}), (-, *)) \to ((-), (-)) \end{array}$$

Each node outputs L when it holds a ♟, otherwise it outputs N.

To establish the correctness of a self-stabilizing algorithm, we define a notion of "safe configuration" and prove two things:

1. Starting from an arbitrary configuration, a safe configuration will eventually be reached.
2. Starting from an arbitrary *safe* configuration, the output traces of all possible executions have a suffix in the desired behavior.

For Algorithm 1, the desired behavior is the leader-election behavior LE, and the *safe configurations* are those in which at least one agent outputs L.

Lemma 1. *Let E be an execution of Algorithm 1 starting from an arbitrary configuration. Then E contains a safe configuration.*

Proof. Suppose no configuration of E is safe. This means there are no leaders in E, so from some point on, every node receives `false` from the leader detector. By rule 2, the initiator of the next interaction will declare itself a leader, a contradiction. Hence, E contains a safe configuration.

Lemma 2. *Let E be an infinite globally or locally fair execution of Algorithm 1 starting from an arbitrary safe configuration. Then the output trace of E has a suffix in LE.*

Proof. Notice that the only way for the number of leaders to decrease is via rule 1. The number of leaders decreases by one only when two leaders interact, so there is always at least one leader in subsequent configurations. Eventually all agents will receive `true` from the leader detector, after which new leaders cease being generated. By either local or global fairness, every two agents interact with each other infinitely often, so eventually the number of leaders will decrease to one. The last leader cannot disappear, no new leaders are created, and the leader status cannot be transferred to another agent. Hence, the output trace of the suffix of E from this point on is in LE.

Theorem 3. *Given $\Omega?$, Algorithm 1 is a self-stabilizing implementation of the leader-election behavior LE that is correct under both global and local fairness.*

Proof. The correctness of Theorem 3 follows from Lemma 1 and Lemma 2.

5 Leader Election in Rings

The ring is an important network topology. The leader-election problem in rings has been extensively studied in the literature [4, 7, 8, 10, 11]. Most of those results assume local fairness or a similar fairness condition. It has been shown that there is no uniform[1] self-stabilizing leader-election algorithm in anonymous rings under the assumption of local fairness.

We refine these results in two ways. First, we show that uniform leader election in anonymous rings remains impossible under local fairness, even with the help of the leader detector $\Omega?$. Second, we exhibit a uniform self-stabilizing leader election algorithm using $\Omega?$ that works in rings of arbitrary size under the assumption of global fairness. We leave open the question of whether such an algorithm exists without the help of $\Omega?$.

5.1 Impossibility under Local Fairness

Theorem 4. *No uniform leader-election algorithm exists in a ring assuming local fairness, even with the help of leader detector $\Omega?$*

Proof. Assume to the contrary that there is a uniform leader-election algorithm for rings that works under local fairness with the help of $\Omega?$. We consider the type of ring that is the most powerful in terms of computation: The ring is a directed cycle, so each node in an interaction knows whether it is talking to its next or previous neighbor in clockwise order around the ring.

For any $n \geq 2$, we look at two rings: R_1 has n nodes labeled $0, 1, \ldots, n-1$ and R_2 has $2n$ nodes labeled $0, 1, \ldots, 2n-1$. Given a locally fair execution E_1 of R_1, we describe a locally fair execution E_2 of R_2 such that all but finitely many configurations of E_2 have exactly two leaders. Hence, any algorithm with correct leader-election behavior on R_1 fails to have leader-election behavior on R_2, showing that there is no uniform leader-election algorithm.

[1] An algorithm is *uniform* if it works in rings of all sizes.

Intuitively, we regard R_2 as two copies of R_1 spliced together into a single ring, as is shown in Figure 1. Each step of E_1 is applied separately to the two copies of R_1 in R_2. After every such pair of steps, the two copies of R_1 that comprise R_2 will be in the same configuration as each other and as R_1. Hence, if R_1 has one leader, then R_2 has two leaders.

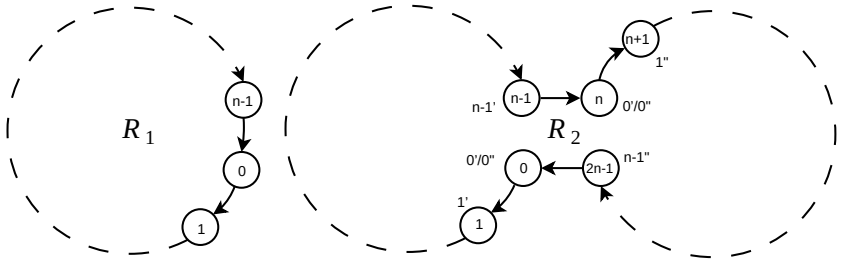

Fig. 1. The rings R_1 and R_2.

Formally, node u of R_1 *corresponds* to the two nodes u and $u+n$ of R_2. An edge $e = (u, v)$ of R_1 *corresponds* to the two edges $e' = (u', v')$ and $e'' = (u'', v'')$ of R_2 such that u corresponds to u' and u'' and v corresponds to v' and v''. For example, if $n = 7$, then $(2, 3)$ of R_1 corresponds to $(2, 3)$ and $(9, 10)$ of R_2, and $(6, 0)$ of R_1 corresponds to $(6, 7)$ and $(13, 0)$ of R_2.[2] We say that configurations C of R_1 and D of R_2 are *compatible* if $C(u) = D(u) = D(u+n)$ for each $u = 0, \ldots, n-1$, i.e., the states of corresponding nodes are the same.

Let $E_1 = (C_0, \alpha_0), (C_1, \alpha_1), \ldots$ be a fair execution of R_1. Let D_0 be a configuration of R_2 that is compatible with C_0. We construct an execution

$$E_2 = (D_0, \alpha_0), (D'_0, \alpha_0), (D_1, \alpha_1), (D'_1, \alpha_1), \ldots$$

of R_2 such that C_t and D_t are compatible for all t. We then argue that E_2 is locally fair and satisfies $\Omega?$, from which we derive a contradiction to the assumption that a uniform leader-election algorithm exists.

At each stage t of the construction, we assume that C_t and D_t are compatible. Let $\sigma_t = (r_t, e_t)$ be an action such that $(C_t, \alpha_t) \xrightarrow{\sigma_t} C_{t+1}$, where r_t is a transition of δ and $e_t = (u_t, v_t)$ is an edge of R_1. Let $\sigma'_t = (r, e')$ and $\sigma''_t = (r, e'')$ be actions, where $e'_t = (u'_t, v'_t)$ and $e''_t = (u''_t, v''_t)$ are the two edges of R_2 that correspond to e_t. Both σ'_t and σ''_t are enabled in D_t. This is because σ_t is enabled in C_t, and since C_t is compatible with D_t, $D_t(u'_t) = D_t(u''_t) = C_t(u_t)$ and $D_t(v'_t) = D_t(v''_t) = C_t(v_t)$.

Let D'_t be the unique configuration such that $(D_t, \alpha_t) \xrightarrow{\sigma'_t} D'_t$. Because $n \geq 2$, the nodes u', u'', v', v'' are all distinct, so the states of u'' and v'' are the same in D_t and in D'_t. Hence, σ''_t is also enabled in D'_t. Let D_{t+1} be the unique

[2] Note that $(6, 0)$ does not correspond to pairs $(6, 0)$ and $(13, 7)$ obtained by interchanging the second components since these latter pairs are not edges of R_2.

configuration such that $(D'_t, \alpha_t) \xrightarrow{\sigma''_t} D_{t+1}$. It is easily shown that C_{t+1} and D_{t+1} are compatible. By induction, C_t and D_t are compatible for all t.

It remains to show that E_2 is a locally fair execution of R_2. It is obviously an execution (since each configuration follows from the previous one by a legal action). We must argue that it is locally fair and that the inputs are consistent with $\Omega?$. Local fairness follows from the correspondence between steps of R_2 and R_1. If some action σ' of R_2 is infinitely often enabled in E_2, then the corresponding action σ of R_1 is infinitely often enabled in E_1. By local fairness of E_1, σ is taken infinitely often in E_1. By the above construction, σ' is taken infinitely often in E_2.

Finally, we argue that the inputs in E_2 satisfy the conditions for $\Omega?$. For each t, if configurations C_t and C_{t+1} both have leaders, then D_t, D'_t, and D_{t+1} all have leaders. Similarly, if C_t and C_{t+1} both lack leaders, then D_t, D'_t, and D_{t+1} all lack leaders. Hence, if all but finitely many configurations of E_1 lack leaders, then all but finitely many configurations of E_2 lack leaders, and similarly, if all but finitely many configurations of E_1 have leaders, then all but finitely many configurations of E_2 have leaders. Hence, the sequence of input assignments $\alpha_0, \alpha_0, \alpha_1, \alpha_1, \ldots$ in E_2 is correct for $\Omega?$.

It follows that E_2 is a locally fair execution of R_2 with leader detector $\Omega?$. However, the output trace of E_2 is not in LE since all but finitely many configurations of E_2 have two leaders. Thus, the assumed algorithm is not a uniform leader-election algorithm.

5.2 Leader Election under Global Fairness

A non-uniform self-stabilizing leader-election algorithm assuming global fairness was given in [4]. Here we give a uniform algorithm with the help of $\Omega?$.

We assume that the ring is *directed*, which means each node has a sense of "forward" (clockwise) and "backward" (counter-clockwise), and every interaction takes place between the initiator and its forward neighbor. A self-stabilizing algorithm to direct an undirected ring was given in [4], so our algorithm can be applied to any weakly connected cycle, whether directed or not.

Each node can store zero or one of each of three kinds of tokens: a bullet "◀", a leader mark "♟", and a shield "▌", for a total of eight possible states. Corresponding to each kind of token is a *slot* which is *empty* if the corresponding token is not present, and *full* if it is present. An empty slot is denoted by "−"; a full slot is denoted by the corresponding token. The slots in each node are ordered with the bullet first, leader mark second, and shield third. Extending this to a clockwise ordering of all slots in the ring, the shield slot of one node is followed by the bullet slot of the next node in clockwise order.

Algorithm 2

Rule 1. $((*\,*\,*, \mathsf{F}), (*\,*\,*, *)) \to ((\blacktriangleleft\, \bullet\, \mathbf{I}), (*\,*\,*))$
Rule 2. $((*\,-\,\mathbf{I}, \mathsf{T}), (*\,*\,*, *)) \to ((*\,-\,-), (-\,*\,\mathbf{I}))$
Rule 3. $((*\,\bullet\,\mathbf{I}, \mathsf{T}), (*\,*\,*, *)) \to ((\blacktriangleleft\, \bullet\, -), (-\,*\,\mathbf{I}))$
Rule 4. $((*\,\bullet\,-, \mathsf{T}), (-\,*\,*, *)) \to ((\blacktriangleleft\, \bullet\, -), (-\,*\,*))$
Rule 5. $((*\,*\,-, \mathsf{T}), (\blacktriangleleft\, *\,*, *)) \to ((\blacktriangleleft - -), (-\,*\,*))$

Each node outputs L when it holds a •*, otherwise it outputs N.*

When two nodes interact and the initiator's input is `false` (F), a leader and a shield are created. At the same time, a bullet is fired (rule 1). This is the only way for leaders and shields to be created. When the initiator's input is `true` (T), the following rules apply: Shields move forward around the ring (rules 2 and 3), and bullets move backward (rule 5). Bullets are absorbed by any shield they encounter (rules 2 and 3) but kill any leaders along the way (rule 5). If a bullet moves into a node already containing a bullet, the two bullets merge into one. Similarly, when two shields meet, they merge into one. A leader fires a bullet whenever it is the initiator of an interaction (rules 3 and 4).

A node i in a configuration is called a *protected leader*, and node j is called its *protecting shield*, if i has a leader mark, j has a shield, and all of the slots between i's leader mark and j's shield in clockwise order are empty. A node can be both a protected leader and its own protecting shield. We show that eventually there is exactly one protected leader, one protecting shield, and no unprotected leader.

Let E be an execution. Define S_E to be the maximal suffix of E such that every (configuration, input assignment) pair in S_E occurs infinitely often. S_E is well-defined and infinite since there are only finitely many distinct (configuration, input assignment) pairs. Define IRC_E[3] to be the set of configurations that occur in S_E.

The follows lemmas are all qualified by "for any execution E".

Lemma 3. *If any configuration in* IRC_E *has a protected leader, then every configuration in* IRC_E *has a protected leader.*

Proof. Let $C \in \mathrm{IRC}_E$ be a configuration with a protected leader, and suppose $(C, \alpha) \to C'$. We show that C' has a protected leader, regardless of which transition rule was applied.

- Rule 1 creates a new protected leader.
- Rule 2 moves the shield forward. If the responder is a leader, it becomes protected. If not, the protected leader is still protected after the move.
- Rule 3 fires a bullet and moves the shield forward. If the responder is a leader, it becomes protected. If not, the initiator remains a protected leader.

[3] IRC stands for Infinitely Recurring Configurations.

- Rule 4 fires a bullet. This does not affect the protected leader, for the bullet cannot be the only non-empty slot between the protected leader and its protecting shield.
- Rule 5 moves a bullet from the responder to the initiator. If the initiator is a leader, it is killed by the bullet. However, the initiator was not protected beforehand since there was no shield between the leader mark and the bullet token. This rule does not affect the protected status of any other leader.

Thus, every configuration in S_E following the first one having a protected leader also has a protected leader. Because every pair in S_E occurs infinitely often, all configurations in IRC_E have a protected leader.

Let α_{F} and α_{T} be input assignments such that α_{F} assigns `false` and α_{T} assigns `true` to every node.

Lemma 4. *If no configuration in* IRC_E *has a leader, then every input assignment in* S_E *is* α_{F}*. If every configuration in* IRC_E *has a leader, then every input assignment in* S_E *is* α_{T}*.*

Proof. Immediate from the definition of leader detector and the fact that every pair in S_E occurs infinitely often.

Lemma 5. *Every configuration in* IRC_E *has at least one leader.*

Proof. Suppose some configuration $C \in \mathrm{IRC}_E$ lacks a leader. There are two cases: If no configuration in IRC_E has a leader, then by Lemma 4, every input assignment in S_E is α_{F}, so every step in S_E is via rule 1. On the other hand, if some configuration $C' \in \mathrm{IRC}_E$ has a leader, then there is a sequence of steps in S_E that goes from (C, α) to (C', α') for some input assignments α and α' since both C and C' occur infinitely often in S_E. One of the steps must be via rule 1 since it is the only leader-creating rule. In either case, the application of rule 1 creates a configuration with a *protected* leader. Lemma 3 then implies that all configurations in IRC_E have protected leaders, contradicting the assumption that C lacks a leader.

Lemma 6. *Every input assignment in* S_E *is* α_{T}*.*

Proof. By Lemma 5, every configuration in IRC_E has a leader. The result follows from Lemma 4.

Lemma 7. *Suppose* $C \in \mathrm{IRC}_E$*,* $C = C_0$*,* C_1*, ...,* $C_r = C'$ *are configurations, and* $(C_i, \alpha_{\mathsf{T}}) \to C_{i+1}$*, for* $i = 0, \ldots, r-1$*. Then* $C' \in \mathrm{IRC}_E$*.*

Proof. An easy induction shows that each $C_i \in \mathrm{IRC}_E$. Suppose $C_i \in \mathrm{IRC}_E$. By Lemma 6, every pair in S_E has input assignment α_{T}. Since $(C_i, \alpha_{\mathsf{T}})$ occurs infinitely often in S_E, $C_{i+1} \in \mathrm{IRC}_E$ by global fairness.

Lemma 8. *Every configuration in* IRC_E *contains the same number of leaders and the same number of shields.*

Proof. By Lemma 6, every input assignment in S_E is α_{T}, so rule 1 is never applied in S_E. Therefore, no step can increase the number n of leaders or the number m of shields. But also, no step can decrease n or m since otherwise S_E would contain only finitely many configurations with n leaders or m shields, a contradiction to the definition of S_E. Hence, no step changes n or m, so all configurations have the same number of leaders and the same number of shields.

Lemma 9. *No configuration in* IRC_E *contains an unprotected leader.*

Proof. Suppose $C \in \mathrm{IRC}_E$ contains an unprotected leader. By Lemma 6, (C, α_{T}) occurs in S_E. From (C, α_{T}), there exists a finite sequence of steps to kill the unprotected leader by applying the rules 3, 4, and 5. By Lemma 7, the resulting configuration C' is in IRC_E. By Lemma 6, no step in S_E can create a new leader, so C' has fewer leaders than C. This contradicts Lemma 8.

Lemma 10. *Every configuration in* IRC_E *contains exactly one shield and exactly one leader.*

Proof. By Lemmas 5 and 9, every configuration in IRC_E contains at least one protected leader. This implies that each configuration contains at least one shield. There must be exactly one, for if any configuration has two or more shields, there exists a finite sequence of steps to merge two shields by applying rules 2 and 3 (possible by Lemma 6), resulting in a configuration C' with fewer shields. By Lemma 7, $C' \in \mathrm{IRC}_E$, contradicting Lemma 8. Finally, each shield is the protecting shield of at most one leader, so each configuration contains exactly one leader.

Theorem 5. *Given* $\Omega?$*, Algorithm 2 is a self-stabilizing implementation of the leader-election behavior* LE *in rings under global fairness.*

Proof. By Lemma 10, every configuration in IRC_E has exactly one leader. The *same* node is leader in every such configuration since none of the five rules can move the leader mark from one node to another in a single step. Hence, $OT(S_E) \in LE$, and Theorem 5 follows.

Algorithm 2 is at the same time a self-stabilizing token-circulation algorithm. After an execution stabilizes, there is exactly one shield moving around the ring, which could provide a token-circulation service.

6 Conclusion

We study the problem of self-stabilizing leader election in a model of finite-state anonymous agents. We consider this problem under two fairness conditions and with two interaction graph topologies. Our protocols utilize a leader detector $\Omega?$ that eventually correctly detects the presence or absence of a leader in the network. We show that the difficulty of leader election in the population-protocol model is due to the difficulty of detecting the presence and absence of leaders. It is an open problem for future research whether $\Omega?$ can be implemented in rings and other families of network graphs in the population-protocol model.

References

1. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **20**(4) (1996) 685 – 722
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Twenty-Third ACM Symposium on Principles of Distributed Computing. (2004) 290–299
3. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In Prasanna, V.K., Iyengar, S., Spirakis, P., Welsh, M., eds.: Distributed Computing in Sensor Systems: First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USE, June/July, 2005, Proceedings. Volume 3560 of Lecture Notes in Computer Science., Springer-Verlag (2005) 63–74
4. Angluin, D., Aspnes, J., Fischer, M.J., Jiang, H.: Self-stabilizing population protocols. In: Ninth International Conference on Principles of Distributed Systems. (2005) 79–90
5. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17**(11) (1974) 643–644
6. Schneider, M.: Self-stabilization. ACM Computing Surveys **25**(1) (1993) 45–67
7. Itkis, G., Lin, C., Simon, J.: Deterministic, constant space, self-stabilizing leader election on uniform rings. In: Workshop on Distributed Algorithms. (1995) 288–302
8. Higham, L., Myers, S.: Self-stabilizing token circulation on anonymous message passing rings. Technical report, University of Calgary (1999)
9. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 424–440
10. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: Eighteenth ACM Symposium on Principles of Distributed Computing. (1999) 199–207
11. Ghosh, S., Gupta, A.: An exercise in fault-containment: Self-stabilizing leader election. Information Processing Letters (59) (1996) 281–288
12. Fernández, A., Jiménez, E., Raynal, M.: Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In: 2006 International Conference on Dependable Systems and Networks. (2006)
13. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
14. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the Twenty-third ACM Symposium on Principles of Distributed Computing. (2004) 328–337