A Space Optimal, Deterministic, Self-Stabilizing, Leader Election Algorithm for Unidirectional Rings

Faith E. Fich[1] and Colette Johnen[2]

[1] Department of Computer Science, University of Toronto, Canada
fich@cs.toronto.edu
[2] Laboratoire de Recherche en Informatique, CNRS–Université de Paris-Sud, France
colette@lri.fr

Abstract. A new, self-stabilizing algorithm for electing a leader on a unidirectional ring of prime size is presented for the composite atomicity model with a centralized daemon. Its space complexity is optimal to within a small additive constant number of bits per processor, significantly improving previous self-stabilizing algorithms for this problem. In other models or when the ring size is composite, no deterministic solutions exist, because it is impossible to break symmetry.

1 Introduction

Electing a leader on a ring is a well studied problem in the theory of distributed computing, with recent textbooks devoting entire chapters to it [19, 3]. It requires exactly one processor in the ring be chosen as a leader. More formally, there is a distinguished subset of possible processor states in which a processor is considered to be a leader. The state of the processor that is chosen leader reaches and then remains within the subset, whereas the states of all other processors remain outside the subset. A related problem of interest is token circulation, where a single token moves around the ring from processor to processor, with at most one processor having the token in any configuration. The formal definition of a processor having a token is that the state of the processor belongs to a subset of distinguished states.

Self-stabilizing algorithms are those which eventually achieve a desired property (for example, having a unique leader) no matter which configuration they are started in (and, hence, after transient faults occur). Dijkstra introduced the concept of self-stabilization and gave a number of self-stabilizing algorithms for token circulation on a bidirectional ring, assuming the existence of a leader [9]. One of these algorithms uses only 3 states per processor [11]. On a unidirectional ring with a leader, Gouda and Haddix [13] can perform self-stabilizing token circulation using 8 states per processor.

Conversely, given a self-stabilizing token circulation algorithm on a ring of size n, there is an easy self-stabilizing algorithm to elect a leader using only an additional $\lceil \log_2 n \rceil$ bits per processor. Specifically, each processor stores a name

from $\{0, \ldots, n-1\}$ as part of its state. Whenever a processor gets the token, it updates its name by adding 1 to the name of its left neighbour and then taking the result modulo n. Eventually, there will a unique processor with name 0, which is the leader.

Without the ability to break symmetry, deterministic self-stabilizing leader election and token circulation are impossible [2]. For example, consider a synchronous system of anonymous processors. If all processors start in the same state with the same environment, they will always remain in the same state as one another. Similarly, in an asynchronous shared memory system of anonymous processors with atomic reads and writes, where all registers have the same initial contents, or in an asynchronous message passing system of anonymous processors, where all communication links contain the same nonempty sequence of messages, many schedules, for example, a round robin schedule, will maintain symmetry among all the processors. Therefore, the study of deterministic algorithms for leader election and token passing in systems of anonymous processors has focussed on Dijkstra's composite atomicity model with a centralized daemon (where a step consists of a state transition by a single processor, based on its state and the states of its neighbours). Even in this model, symmetry among equally spaced, nonadjacent processors in a ring can be maintained by an adversarial scheduler. Therefore, deterministic algorithms for leader election and token circulation are possible in a ring of n anonymous processors only when n is prime [2, 10, 7].

Randomization is a well known technique to break symmetry and randomized algorithms for both problems have been considered on a variety of models [1, 15, 4]. This work is beyond the scope of our paper.

There are deterministic self-stabilizing leader election algorithms for bidirectional rings (of prime size using the composite atomicity model with a centralized daemon) that use only a constant amount of space per processor [17]. For unidirectional rings, Burns and Pachl [7] presented a deterministic self-stabilizing token circulation algorithm that uses $O(n^2)$ states per processor, as well as a more complicated variant that uses $O(n^2/\log n)$ states per processor. They left the determination of the space complexity of this problem as an open question. Lin and Simon [18] further improved their algorithm to $O\left(n\sqrt{n/\log n \log\log n}\right)$ states per processor.

Beauquier, Gradinariu, and Johnen [4] proved a lower bound of n states per processor for any deterministic self-stabilizing leader election algorithm on a unidirectional ring of size n. They also mentioned a similar lower bound of $(n-1)/2$ states per processor for token circulation due to Jaap-Henk Hoepman.

In Section 3, we present a deterministic self-stabilizing leader election algorithm for unidirectional rings (of prime size using the composite atomicity model with a centralized daemon) in which the number of states is $O(n)$. This matches the lower bound to within a small constant factor. Hence, our algorithm matches the number of bits of storage used at each processor to within a small additive constant of the number required by the lower bound. An algorithm for self-stabilizing token circulation on a unidirectional ring can be obtained by

combining our algorithm for electing a leader with Gouda and Haddix's token circulation algorithm that assumes the existence of a leader [13]. The resulting algorithm has provably optimal space complexity to within a small additive constant, solving Burns and Pachl's open question.

Our algorithm was inspired by and is closely related to Burns and Pachl's basic algorithm. To achieve small space, our idea is to time share the space: two pieces of information are stored alternately in one variable instead of in parallel using two different variables. However, the correct implementation of this simple idea in a self-stabilizing manner is non-trivial.

When developing our algorithm, we use Beauquier, Gradinariu, and Johnen's alternating schedule approach [4, 5] to simplify the description and the proof of correctness. In Section 2, we give a more careful description of our model of computation, define the set of alternating schedules, and state some important properties of executions that have alternating schedules. Most of these results describe how information flows from one processor to another during the course of an execution.

2 The Model

We consider a system consisting of n identical, anonymous processors arranged in a ring, where n is prime. The value of n is known to the processors. The left neighbour of a processor P will be denoted P_L and its right neighbour will be denoted P_R. The ring is *unidirectional*, that is, each processor can only directly get information from its left neighbour. The *distance* from processor P to processor Q is measured starting from P and moving to the right until Q is reached. In particular, the distance from P to P_L is $n - 1$.

In any algorithm, each processor is in one of a finite number of states. A *configuration* specifies the state of every processor. An *action* of a processor is a state transition, where its next state depends on its current state and the state of its left neighbour. Note that the next state might be the same as the current state. Only one processor performs an action at a time. This is Dijkstra's composite atomicity model with a centralized daemon [9]. An algorithm is *deterministic* if, for each processor, its actions can be described by a total state transition function from the cross product of its state set and the state set of its left neighbour. In other words, in every configuration, each processor has exactly one action it can perform. A processor is *enabled* in a configuration if there is an action that causes the processor to change its state. Some authors prefer to describe a deterministic algorithm using partial state transition functions, not defining those transitions in which a processor is not enabled.

A *schedule* is a sequence whose elements are chosen from the set of n processors. If P is the t'th element of the schedule, we say that t *is a step of* P and P *takes a step at time* t. A processor P *takes a step during the time interval* $[t, t'']$ if it takes a step at some time t', where $t \leq t' \leq t''$. In particular, if $t > t''$, then the interval $[t, t'']$ is empty and no processor takes a step during this interval.

An *execution* is an infinite sequence of configurations and processors

$$\mathit{config}(0), \mathit{proc}(1), \mathit{config}(1), \mathit{proc}(2), \mathit{config}(2), \ldots$$

where configuration $\mathit{config}(t)$ is obtained from configuration $\mathit{config}(t-1)$ by the action of processor $\mathit{proc}(t)$, for all $t > 0$. We say that $\mathit{config}(t)$ is the configuration at time t of the execution. The initial configuration of this execution is $\mathit{config}(0)$, the configuration at time 0. The *schedule of the execution* is the subsequence $\mathit{proc}(1), \mathit{proc}(2), \ldots$ of processors. An infinite schedule or execution is *fair* if every processor appears in the sequence infinitely often.

Fix an execution. If processor P takes a step at time T, then the state of P at time T is influenced by the state of P_L at time $T-1$. If P does not take any steps in the interval $[T+1, t']$, then it will have the same state at T and t'. Similarly, if P_L does not take any steps in the interval $[t+1, T]$, then it will have the same state at t and $T-1$. Thus the state of P_L at time t influences the state of P at time t'. The following definition extends this relationship to pairs of processors that are further apart.

Definition 1. *Suppose $P_0, P_1, \ldots, P_k$ are $k+1 \leq n$ consecutive processors, in order, rightwards along the ring. Then the state of P_0 at time t_0* **influences** *the state of P_k at time $t' > t_0$, denoted*

$$(P_0, t_0) \rightarrow (P_k, t'),$$

if and only if there exist times $t_0 < t_1 < \cdots < t_k \leq t'$ such that P_k takes no steps during the time interval $[t_k+1, t']$ and, for $i = 1, \ldots, k$, P_{i-1} takes no steps during the time interval $[t_{i-1}+1, t_i]$, but P_i takes a step at time t_i.

This definition of influence only captures the communication of information around the ring. It does not capture knowledge that a processor retains when it takes steps. For example, if P takes a step at time T, then $(P, T-1) \not\rightarrow (P, T)$. The following results are easy consequences of the definition.

Proposition 1. *Suppose P, P', and P'' are distinct processors with P' on the path from P to P''. If $(P,t) \rightarrow (P'', t'')$ then there exists $t < t' < t''$ such that P' takes a step at time t', $(P,t) \rightarrow (P', t')$, and $(P', t') \rightarrow (P'', t'')$. Conversely, if $(P,t) \rightarrow (P', t')$ and $(P', t') \rightarrow (P'', t'')$, then $(P,t) \rightarrow (P'', t'')$.*

Proposition 2. *Suppose P takes no steps in the interval $[t+1, T]$ and P' takes no steps in the interval $[t'+1, T']$, where $t \leq T$ and $t' \leq T'$. Then the following are equivalent: $(P,t) \rightarrow (P', t')$, $(P,t) \rightarrow (P', T')$, $(P,T) \rightarrow (P', t')$, and $(P,T) \rightarrow (P', T')$.*

A subset of the configurations of an algorithm is *closed* if any action performed from a configuration in this set results in a configuration in this set. Let H be a predicate defined on configurations. An algorithm *stabilizes to H under a set of schedules S* if there is a closed set of configurations, L, all of which satisfy H, such that every execution whose schedule is in S contains a configuration

that is in L. The configurations in L are called *safe*. When an execution reaches a safe configuration, we say that it has stabilized. The *stabilization time* of an algorithm is the maximum, over all executions in S, of the number of actions performed until the execution stabilizes. A self-stabilizing algorithm is *silent* if no processors are enabled in safe configurations.

Let LE be the predicate, defined on configurations of an algorithm, that is true when exactly one processor is a leader (i.e. its state is in the specified set). An algorithm that stabilizes to LE under the set of all fair schedules is called a *self-stabilizing leader election algorithm*. Notice that, once an execution of a leader election algorithm stabilizes, the leader does not change. This is because processors change state one at a time, so between a configuration in which one processor is the only leader and a configuration in which another processor is the only leader, there must be an unsafe configuration.

We present an algorithm in Section 3 that stabilizes to LE under the set of alternating schedules, defined in Section 2.1, using $5n$ states per processor. Beauquier, Gradinariu, and Johnen [4] prove the following result about stabilization under the set of alternating schedules.

Theorem 1. *Any algorithm on a ring that stabilizes to predicate H under the set of alternating schedules can be converted into an algorithm that stabilizes to H under all fair schedules, using only double the number of states (i.e. only one additional bit of storage) at each processor.*

Applying their transformation to our algorithm gives a self-stabilizing leader election algorithm that uses $10n$ states per processor.

2.1 Alternating Schedules

A schedule is *alternating* if, between every two successive steps of each processor, there is exactly one step of its left neighbour and exactly one step of its right neighbour. Any round robin schedule is alternating. For a ring of size 5 with processors P_1, P_2, P_3, P_4, P_5 in order around the ring, the finite schedule $P_1, P_2, P_5, P_4, P_1, P_5, P_3, P_4, P_2, P_3, P_1, P_2, P_5, P_1, P_4, P_5, P_3, P_2$ is also an alternating schedule. It is equivalent to say that a schedule is alternating if, between every two steps of each processor, there is at least one step of each of its neighbours.

The assumption of an alternating schedule allows us to determine more situations where the state of a processor at one step influences the state of another processor at some later step. The proofs of the following lemmas are by induction on the distance from P to Q and can be found in the complete paper. They will be used in the proof that our algorithm stabilizes to LE under the set of alternating schedules.

The first result says that if a step of P influences a step of Q, then earlier or later steps of P influence correspondingly earlier or later steps of Q.

Lemma 1. *Consider an alternating schedule in which processor P takes k steps in the interval $[t+1, T]$ and processor Q takes k steps in the interval $[t'+1, T']$. Then $(P, t) \rightarrow (Q, t')$ if and only if $(P, T) \rightarrow (Q, T')$.*

Another important property is that each step of each processor will eventually influence some step of every other processor.

Lemma 2. *Let P and Q be distinct processors. Suppose P takes a step at time t of an alternating schedule. Then there exists a step $t' > t$ of Q such that $(P, t) \rightarrow (Q, t')$.*

The next results bound when a processor will influence another processor, as a function of the distance from one to the other.

Lemma 3. *Let P and Q be distinct processors, where the distance from P to Q is k. If P takes at least $k + 1$ steps by time t' in an alternating schedule, then there exists a time $t < t'$ such that $(P, t) \rightarrow (Q, t')$, P takes at most k steps in the interval $[t + 1, t']$, and P takes a step at time t.*

Lemma 4. *Let P and Q be distinct processors, where the distance from P to Q is k. If Q takes at least k steps by time t' in an alternating schedule, then there exists a time $t < t'$ such that $(P, t) \rightarrow (Q, t')$, Q takes at most k steps in the interval $[t, t']$, and either $t = 0$ or P takes a step at time t.*

Finally, the difference between the numbers of steps that have been taken by two processors can be bounded by the distance from one processor to the other.

Lemma 5. *Suppose the distance from P to Q is k and $(P, t) \rightarrow (Q, t')$. If Q takes at least m steps by time t' in an alternating schedule, then P takes at least $m - k$ steps by time t.*

Lemma 6. *Suppose the distance between P and Q is k. If P takes m steps by time t, then Q takes between $m - k$ and $m + k$ steps by time t.*

3 A New Leader Election Algorithm

In this section, we present a deterministic leader election algorithm for a unidirectional ring that uses $5n$ states per processor and stabilizes under the set of alternating schedules within $O(n^2)$ time.

We begin by describing some of the ideas from Burns and Pachl's token circulation algorithm [7] and how they are used in our leader election algorithm.

In a token circulation algorithm, some, but not all, of the tokens must disappear when more than one token exists. Similarly, in a leader election algorithm, when a ring contains more than one leader, some, but not all, of the leaders must become nonleaders. Because the only direct flow of information is from a processor to the processor on its right, the first token or leader a processor can receive information from is the first one that is encountered travelling left from the processor. We call this the *preceding* token or leader. The *following* token or leader is the closest one to the processor's right. We define the *strength* of a token or leader to be the distance to it from the preceding token or leader. If there is only one token or leader in a ring of size n, then its strength is n.

In Burns and Pachl's basic algorithm, each processor has two variables: one to store its distance from the preceding token (which, in the case of a processor with a token is the strength of that token) and the other to store the strength of the preceding token. The value of each variable is in $[1, n]$. Thus, the total number of states per processor is $O(n^2)$.

A processor whose left neighbour has a token knows that it is at distance 1 from the preceding token. Any other processor can determine its distance from the preceding token by adding 1 to the corresponding value of its left neighbour. Every processor can obtain the strength of the preceding token directly from its left neighbour: from the first variable, if its left neighbour has a token and from the second variable, if its left neighbour does not have a token.

When a processor with a token learns that the preceding token is stronger, it destroys its own token. On a ring whose size is prime, the distances between successive tokens cannot be all identical. Thus, extra tokens will eventually disappear.

In our algorithm, the state of each processor consists of two components: a *tag* $X \in \{c, d, B, C, D\}$ and a *value* $v \in [1, n]$. We say that a processor is a *leader* if its tag is B, C, or D; otherwise it is called a *nonleader*.

The safe configurations of our algorithm each contain exactly one leader, which is in state (D, n). In addition, the nonleader at distance i from the leader is in state (d, i), for $i = 1, \ldots, n-1$. It is easy to verify that no processors are enabled in safe configurations. Hence, our algorithm is silent.

Actions of our algorithm are described by specifying the state (X, v) of a processor P and the state (X_L, v_L) of its left neighbour P_L and then giving P's new state (X', v'). Such an action will be written

$$(X_L, v_L)\ (X, v) \mapsto (X', v').$$

Instead of presenting all the actions at once, we present them in small groups, together with a brief discussion of their intended effect.

For a leader, v usually contains its *strength*, i.e. its distance from the preceding leader. Sometimes, nonleaders are used to *determine* the strength of a leader. In this case, the processor has tag d and v contains its *distance* from the preceding leader. A nonleader can also have tag c, indicating that v is *conveying* strength information from the preceding leader to the following leader.

An example of an unsafe configuration of our algorithm is illustrated in Figure 1. Here P and P' are leaders with strength 4, P'' is a leader with strength 3, and the other 8 processors are nonleaders.

When the tag of a leader is B or D, this signals the sequence of nonleaders to its right to compute their distance from the leader, as follows: The right neighbour of the leader sets its value to 1 and each subsequent nonleader sets its value to one more than the value of the nonleader to its left. They set their tags to d, to relay the signal. Because the schedule is alternating, the signal is guaranteed to reach the entire sequence of nonleaders to the right of the leader.

1. $(X_L, v_L)\ \ (X, v)\ \mapsto (d, 1)$ for $X_L = B, D$ and $X = c, d$

2. $(d, v_L)\ \ (X, v)\ \mapsto (d, 1 + (v_L \bmod n))$ for $X = c, d$ and $v_L \neq n - 1$

When the tag of a leader is C, this signals the nonleaders to its right to convey the strength of this leader, by copying the value from their left neighbour and setting their tag to c. For example, in Figure 1, if the processor at distance 2 from P'' takes the next step, its state will change from $(d, 2)$ to $(c, 3)$.

3. $(X_L, v_L)\ \ (d, v)\ \mapsto (c, v_L)$ for $X_L = c, C$

Our algorithm ensures that if a processor has tag C or c immediately before it takes a step, then it has neither tag immediately afterwards. This implies that processors with tag c will not perform either of the following two actions after taking their first step. When the tag of its left neighbour is C, processor P with tag c simply treats its left neighbour's tag as if it were D and enters state $(d, 1)$. However, when its left neighbour's tag is c, it is possible that all processors have tag c. To ensure that the ring contains at least one leader, P becomes a leader.

4. $(C, v_L)\ \ (c, v)\ \mapsto (d, 1)$
5. $(c, v_L)\ \ (c, v)\ \mapsto (B, 1)$

A processor in state $(d, n - 1)$ suggests that its right neighbour is the only leader. If that right neighbour is a nonleader, it can correct the problem by becoming a leader. This avoids another situation where no leader may exits.

6. $(d, n - 1)\ \ (X, v)\ \mapsto (B, 1)$ for $X = c, d$

A leader with tag B is a *beginner*: it has performed n or fewer steps as a leader. For beginners, the value v records the number of steps for which the processor has been a leader, up to a maximum of n. Thus, when a nonleader becomes a leader, it begins in state $(B, 1)$. After a leader has performed n steps as a beginner, it should get tag D and record its strength in v. But when its left neighbour has tag c, the processor cannot determine its own strength. Therefore it waits in state (B, n) until its next step. In the meanwhile, its left neighbour will take exactly one step and, hence, will have a different tag. Thus, a processor performs at most $n + 1$ consecutive steps as a beginner.

7. $(X_L, v_L)\ \ (B, v)\ \mapsto (B, v + 1)$ for $X_L = B, c, d$ and $v \neq n$
8. $(d, v_L)\ \ (B, n)\ \mapsto (D, 1 + (v_L \bmod n))$
9. $(B, v_L)\ \ (B, n)\ \mapsto (D, 1)$
10. $(c, v_L)\ \ (B, n)\ \mapsto (B, n)$

When the system is in a configuration with multiple leaders, some of these leaders have to be destroyed. If P is a leader whose left neighbour has tag C or D, then P resigns its leadership by setting its state to $(d, 1)$. However, if P's left neighbour has tag B, then P waits in state $(D, 1)$. Provided P's left neighbour stays a leader long enough, P will be destroyed, too.

11. $(X_L, v_L)\ \ (X, v)\ \mapsto (d, 1)$ for $X_L = C, D$ and $X = B, C, D$

12. $(B, v_L)\ \ (X, v)\ \mapsto (D, 1)$ for $X = C, D$

When P is a leader whose left neighbour, P_L, has tag c, then v_L contains the strength of the preceding leader. In this case, P can compare its strength against that of the preceding leader. If P is stonger, it remains a leader. Otherwise, P resigns its leadership by setting its tag to d. However, the value v_L does not provide information from which P can compute its distance from the preceding leader. Consequently, P sets its value to n, to act as a place holder until P's next step, when P_L will have a different tag.

13. $(c, v_L)\ \ (X, v)\ \mapsto (D, v)$ for $X = C, D$ and $v \geq v_L$
14. $(c, v_L)\ \ (X, v)\ \mapsto (d, n)$ for $X = C, D$ and $v < v_L$

A processor can enter state (d, n) only by resigning its leadership. When the left neighbour of a leader is in state (d, n), the leader cannot use v_L to determine its strength. In this case, the leader leaves its value v unchanged.

15. $(d, n)\ \ (D, v)\ \mapsto (D, v)$

When its left neighbour P_L has tag d, a leader P that is not a beginner can update its value with a better estimate of its strength. Such a processor usually alternates its tag between C and D. The only exception is when P's left neighbour has value $n-1$, which indicates that P is the only leader. In this case, P stays in state (D, n).

16. $(d, v_L)\ \ (D, v)\ \mapsto (C, 1 + v_L)$ for $v_L \neq n-1, n$
17. $(d, v_L)\ \ (C, v)\ \mapsto (D, 1 + (v_L \bmod n))$
18. $(d, n-1)\ \ (D, v)\ \mapsto (D, n)$

4 Properties of the Algorithm

In this section, we present a number of results about the behaviour of our algorithm. They are useful for the proof of correctness presented in Section 5. Throughout this section and Section 5, we assume that all schedules are alternating.

The first result relates the value of a processor with tag d to its distance from a leader or a processor that has just resigned its leadership. It can be proved by induction.

Lemma 7. *Suppose $(P, t) \to (Q, t')$, the distance from P to Q is k, and Q has tag d at time t'. If, at time t, P is a leader or has state (d, n), then, at time t', either Q has value n or value at most k. Conversely, if Q has value k at time t', then, at time t, either P is a leader or has state (d, n).*

A leader that is not a beginner cannot have become a leader recently.

Lemma 8. *Let $[t+1, t']$ be an interval that contains at most n steps of processor P. If P has tag C or D at time t', then P is a leader throughout the interval $[t, t']$.*

Proof. Suppose that P becomes a leader during the interval $[t+1, t']$. At that time, P has state $(B, 1)$. It cannot get tag C or D until it has performed at least n more steps, which occurs after time t'.

The next result identifies a situation in which a leader cannot be created.

Lemma 9. *Suppose $(P, t) \rightarrow (Q, t')$ and Q takes at least one step before t'. If P is a leader throughout the interval $[t, t'-1]$, then Q does not become a leader at time t'.*

Proof sketch. Suppose Q becomes a leader at time t'. It follows that Q_L has state $(d, n-1)$ at time $t'-1$. Then Lemma 7 is applied to obtain a contradiction.

4.1 Experienced Leaders

An experienced leader is a processor that has been a leader long enough so that the fact that it is a leader influences and has been influenced by every other processor. Formally, an *experienced leader* is a processor with tag C or D that has taken at least n steps. Then, either an experienced leader has remained a leader since the initial configuration, or it has served its full time as a beginner, since last becoming a leader.

The existence of an experienced leader in a configuration of an execution provides a lot of information about what actions may occur.

Lemma 10. *No new leader will be created whenever there is an experienced leader.*

Proof. To obtain a contradiction, suppose there is a time t'' at which P is an experienced leader and Q becomes a leader. Since P has taken at least n steps, it follows by Lemma 6 that Q has taken at least one step before time t''. This implies that, at time t'', Q performs action 6, Q_L has state $(d, n-1)$, and $P \neq Q, Q_L$.

From Lemma 3 and Proposition 1, there are times $t < t' < t''$ such that $(P, t) \rightarrow (Q_L, t') \rightarrow (Q, t'')$, Q_L takes a step at time t', and P takes at most n steps in the interval $[t, t'']$. Then Lemma 8 implies that P is a leader throughout this interval. Since the distance from P to Q_L is at most $n-2$, it follows from Lemma 7 that Q_L cannot have value $n-1$ at time t', However, Q_L has state $(d, n-1)$ at time t'' and takes no steps in the interval $[t'+1, t'']$. Thus Q_L has state $(d, n-1)$ at time t'. This is a contradiction.

The proofs of the next two results appear in the full paper. They use Proposition 2 and Lemmas 1, 4, 3, 5, 7, 8, and 9.

Lemma 11. *If an experienced leader has value $v > 1$, then the $v-1$ processors to its left are nonleaders.*

This says that the value of an experienced leader is a lower bound on its strength.

Lemma 12. *While a processor is an experienced leader, its value never decreases.*

5 Proof of Self-Stabilization

Here, we prove that the algorithm presented in Section 3 stabilizes to LE under the set of alternating schedules.

The proof has the following main steps. First, we show that every execution reaches a configuration in which there is a leader. Then, from some point on, all configurations will contain an experienced leader. By Lemma 10, no new leaders will be created, so, eventually, all leaders will be experienced leaders. As in Burns and Pachl's algorithm, if there is more than one leader, they cannot have the same strength, because the ring size is prime. Thus resignations must take place until only one leader remains. Finally, a safe configuration is reached.

Lemma 13. *Consider any time interval $[t, t']$ during which each processor takes at least n steps. Then there is a time in $[t-1, t']$ at which some processor is leader.*

Proof. Without loss of generality, we may assume that $t = 1$. To obtain a contradiction, suppose that no processor is a leader in $[0, t']$. Only actions 2 and 3 are performed during $[1, t']$, since all other actions either require or create a leader.

Let v be the maximum value that any processor has as a result of performing action 2 for its first time. Then $1 \leq v < n$. Say processor P_0 has value v at time t_0 as a result of performing action 2 for its first time. Let P_i be the processor at distance i from P_0, for $i = 1, \ldots, n - v$. Then by Lemma 2, there exist times $t_1 < \cdots < t_{n-v}$ such that $(P_0, t_0) \rightarrow (P_1, t_1) \rightarrow \cdots \rightarrow (P_{n-v}, t_{n-v})$ and P_i takes a step at time t_i for $i = 1, \ldots, n - v$. Note that P_{n-v} takes at most n steps by time t_{n-v}; otherwise, Lemma 5 implies that P_0 takes at least two steps before t_0. This is impossible, since P_0 cannot perform action 3 twice in a row. Hence, $t_{n-v} \leq t'$.

It follows by induction that processor P_{n-1-v} has state $(d, n-1)$ at time t_{n-1-v}. But then P_{n-v} performs action 6 at time t_{n-v} and becomes a leader. This is a contradiction.

Lemma 14. *If there is only one experienced leader and it has taken at least $3n$ steps, then it cannot resign its leadership.*

Proof sketch. To obtain a contradiction, suppose that processor P_0 has taken at least $3n$ steps before time t_0, it is the only experienced leader at time $t_0 - 1$, and it resigns its leadership at time t_0. Then P_0 performs action 14 at time t_0. Let v_0 denote the value of P_0 at time $t_0 - 1$ and let $k_0 = 0$.

We prove, by induction, that there exist processors $P_1, \ldots, P_{n-1}$, values $v_0 < v_1 < \cdots < v_{n-1}$, distances $1 < k_1 < \cdots < k_{n-1} < n$, and times $t_1, t'_1, \ldots, t_{n-1}, t'_{n-1}$ such that, for $i = 1, \ldots, n-1$,

- P_i performs action 14 at time $t_i < t_0$,
- P_i takes a step at time $t'_i < t_i$,
- the distance from P_i to P_0 is k_i,
- P_i has value v_i at time $t_i - 1$, and

– $(P_i, t'_i) \to (P_0, t_0)$.

But this is impossible.

Lemma 15. *After each processor has taken $6n + 1$ steps, there is always an experienced leader.*

Proof. Consider any execution of the algorithm. Let t be the first time at which every processor has taken at least $3n$ steps and let $t'' > t$ be the first time such that all processors have taken at least n steps in $[t+1, t'']$. By Lemma 13, there is a time $t' \in [t, t'']$ at which some processor P is a leader. If P has tag C or D at time t', then P is an experienced leader. Otherwise, P has state (B, v) for some value $v \geq 1$. Unless an experienced leader is created, P will perform action 7 at each step until it has state (B, n), it will perform action 10 at most once, and then perform action 8 or 9 to become an experienced leader. Note that, at t', every processor has taken at least $3n$ steps, so Lemma 14 implies that there is at least one experienced leader in every subsequent configuration.

By time t, some processor has taken exactly $3n$ steps, so Lemma 6 implies that no processor has taken more than $\lfloor 7n/2 \rfloor$ steps. Similarly, some processor takes exactly n steps in the interval $[t+1, t'']$, so no processor takes more than $\lfloor 3n/2 \rfloor$ steps in this interval. This implies that P takes at most $5n$ steps by time t''. Therefore P becomes an experienced leader within $6n+1$ steps, since P takes at most $n + 1$ steps after t' until this happens.

Lemma 16. *After each processor has taken at least $\lfloor 15n/2 \rfloor + 2$ steps, all leaders are experienced.*

Proof. Consider any execution of the algorithm and let t be the first time at which every processor has taken at least $6n+1$ steps. By Lemma 15, the execution contains an experienced leader at all times from t on. Lemma 10 implies that no new leader will be created after time t. Any beginner at t will become an experienced leader or resign its leadership by the time it has taken $n + 1$ more steps. By time t, some processor has taken exactly $6n + 1$ steps, so Lemma 6 implies that no processor has taken more than $\lfloor 13n/2 \rfloor$ steps. Thus, by the time each processor has taken $\lfloor 15n/2 \rfloor$ steps, all leaders are experienced.

Lemma 17. *Consider any interval $[t, t']$ during which the set of leaders does not change, all leaders are experienced, and every processor performs at least $n + 1$ steps. Then, at t', the value of every leader is equal to its strength.*

Proof. Let Q be a leader with value v at t' and let P be the processor such that the distance from P to Q is v. Suppose T' is the last time at or before t' at which Q takes a step and Q_L has tag d. Processor Q_L has value $v - 1$ at T'. By Lemma 3, there is a time $T < T'$ such that $(P, T) \to (Q_L, T')$, P takes at most $v - 1$ steps in $[T + 1, T']$, and P takes a step at time T. Lemma 7 implies that, at time T, either P is a leader or has state (d, n).

If P has state (d, n) at time T, then P must have performed action 14 at time T, resigning its leadership. But $t < T < T' \leq t'$ and the set of leaders doesn't

change throughout the interval $[t, t']$. Thus P is a leader at time T and, hence, at time t'.

By Lemma 11, the $v-1$ processors to the left of Q are nonleaders at t'. Hence, at t', processor P is the leader preceding Q and Q has strength v.

Lemma 18. *Let t be any time at which there is more than one leader and all leaders are experienced. If every processor takes at least $\lfloor 5n/2 \rfloor + 2$ steps in $[t, t'']$, then some processor resigns during $[t+1, t'']$.*

Proof. To obtain a contradiction, suppose that, during $[t, t'']$, all processors take at least $\lfloor 5n/2 \rfloor + 2$ steps and the set of leaders does not change. Let $t' > t$ be the first time such that every processor performs at least $n+1$ steps in $[t, t']$. Then Lemma 17 implies that, throughout $[t', t'']$, the value of every leader is equal to its strength. Let P be one leader, let P' be the following leader, and let v and v' denote their respective strengths. The processor that takes step t' takes exactly n steps during the interval $[t, t'-1]$. Then Lemma 6 implies that P takes at most $\lfloor 3n/2 \rfloor$ steps during $[t, t']$.

Consider the first time $T \geq t'$ at which P has tag C. Then P performs at most 2 steps in $[t'+1, T]$. By Lemma 2, there exist times $T < T_1 < \cdots < T_{v'-1} < T'$ such that $(P, T) \to (P_1, T_1) \to \cdots \to (P_{v'-1}, T_{v'-1}) \to (P', T')$, where P_i is the processor at distance i from P. At time T_i, processor P_i performs action 3 and gets state (c, v), for $i = 1, \ldots, v'-1$.

From Lemma 3, we know that P takes at most $v' \leq n-1$ steps in $[T+1, T']$. Hence P takes at most $\lfloor 5n/2 \rfloor + 1$ steps during $[t, T']$. Therefore $T' < t''$. Since no processor resigns during $[t+1, t'']$, processor P' performs action 13 at time T'. Therefore $v' \geq v$.

Since P is an arbitrary leader, this implies that the strengths of all leaders are the same. Hence, the ring size n is divisible by the number of leaders. This is impossible, because n is prime and the number of leaders lies strictly between 1 and n.

Lemma 19. *From any configuration in which there is only one leader and that leader is experienced, the algorithm in Section 3 reaches a safe configuration within $O(n^2)$ steps.*

Proof. Consider a time t at which there is only one leader P and suppose P is experienced at t. By Lemmas 2 and 3, there exists a step t' of P_L such that $(P, t) \to (P_L, t')$ and P takes at most n steps in $[t, t']$. If P_L has tag d at time t', let $T' = t'$; otherwise, let T' be the time of P_L's next step. Then P_L has tag d at time T'. By Lemma 1, there exists a time T such that $(P, T) \to (P_L, T')$ and, by Lemma 7, P_L will have value $n-1$ at time T'.

Processor P gets state (D, n) at its first step following T'. From then on, P remains in state (D, n), performing only actions 18 and 13 . Lemma 7 and an easy induction on k show that if the distance from P to Q is k, P has state (D, n) at time t'', and $(P, t'') \to (Q, T'')$, then Q has state (d, k) at time T''. Thus, within $O(n^2)$ steps, a safe configuration is reached.

Our main result follows directly from these lemmas.

Theorem 2. *The algorithm in Section 3 stabilizes to LE within $O(n^3)$ steps under any alternating schedule.*

6 Conclusion

We have presented a deterministic, self-stabilizing leader election algorithm for unidirectional, prime size rings of identical processors, proved it correct, and analyzed its complexity. The number of states used by this algorithm is $10n$ per processor, matching the lower bound [4] to within a small constant factor. Combined with Gouda and Haddix's algorithm [13], we get a deterministic, self-stabilizing algorithm for token circulation on unidirectional prime size rings that uses only a linear number of states per processor. This answers the open question in [7] of determining the space complexity of of self-stabilizing token circulation on unidirectional rings. We believe our work sheds new insight into the nature of self-stabilization by more precisely delineating the boundary between what is achievable and what is not.

Through our work with alternating schedules, we have improved our understanding of how the state of one processor influences the states of other processors. This enabled us to store two pieces of information alternately in a single variable, yet have both pieces of information available when needed. This technique may be useful for designing other space efficient self-stabilizing algorithms on the ring and, more generally, on other network topologies.

Our algorithm is silent under an alternating schedule. However, when combined with the deterministic token algorithm, it is not silent: the deterministic tokens circulate forever. One remaining open question is whether there exists a silent, deterministic, self-stabilizing leader election on a unidirectional ring that uses only a linear number of states per processor.

Acknowledgements

Faith Fich was supported by grants from the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Ontario. Part of this research was done while she was a visitor at Laboratoire de Recherche en Informatique, Université de Paris-Sud.

References

1. K. Abrahamson, A. Adler, R. Gelbart, L. Higham, and D. Kirkpartrick: The bit complexity of randomized leader election on a ring. SIAM J. Comput. **18** (1989) 12–29
2. Dana Angluin: Local and global properties in networks of processors. 12th ACM Symposium on the Theory of Computing. (1980) 82–93
3. Hagit Attiya and Jennifer Welch: Distributed Computing, Fundamentals, Simulations and Advanced Topics. McGraw-Hill. (1998)

4. J. Beauquier, M. Gradinariu, and C. Johnen: Memory space requirements for self-stabilizing leader election protocols. Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing. (1999) 199–208
5. J. Beauquier, M. Gradinariu, and C. Johnen: Self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Internal Report, LRI, Université de Paris-Sud, France. (1999)
6. J. Beauquier, M. Gradinariu, and C. Johnen: Cross-over composition–enforcement of fairness under unfair adversary. Fifth Workshop on Self-Stabilizing Systems. (2001)
7. J.E. Burns and J. Pachl: Uniform self-stabilizing rings. ACM Transactions on Programming Languages and Systems. **11** (1989) 330–344
8. S. Dolev, M.G. Gouda, and M. Schneider: Memory requirements for silent stabilization. Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. (1996) 27–34
9. E.W. Dijkstra: Self stabilizing systems in spite of distributed control. CACM **17** (1974) 643–644
10. E.W. Dijkstra: Self stabilizing systems in spite of distributed control. Selected Writing on Computing: A Personal Perspective. Springer-Verlag. (1982) 41–46
11. E.W. Dijkstra: A belated proof of self-stabilization. Distributed Computing. **1** (1986) 5–6
12. Shlomi Dolev: Self-Stabilization. MIT Press. (2000)
13. M.G. Gouda and F.F. Haddix: The stabilizing token ring in three bits. Journal of Parallel and Distributed Computing. **35** (1996) 43–48
14. Ted Herman: Comprehensive self-stabilization bibliography. http://www.cs.uiowa.edu/ftp/selfstab/bibliography/ (2001)
15. L. Higham and S. Myers: Self-stabilizing token circulation on anonymous message passing. Distributed Computing: OPODIS '98. Hermes. (1998) 115–128
16. S.T. Huang: Leader election in uniform rings. ACM Transactions on Programming Languages and Systems. **15** (1993) 563–573
17. G. Itkis, C. Lin, and J. Simon: Deterministic, constant space, self-stabilizing leader election on uniform rings. Proceedings of the 9th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science. Springer-Verlag. **972** (1995) 288–302
18. C. Lin and J. Simon: Observing self-stabilization. Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing. (1992) 113–123
19. Nancy Lynch: Distributed Algorithms. Morgan Kaufmann. (1996)