Self-Stabilizing Master–Slave Token Circulation and Efficient Size-Computation in a Unidirectional Ring of Arbitrary Size

Wayne Goddard and Pradip K Srimani
School of Computing
Clemson University
Clemson, SC 29634–0974
{goddard, srimani}@cs.clemson.edu

Abstract

Self-stabilizing algorithms represent an extension of distributed algorithms in which nodes of the network have neither coordination, synchronization, nor initialization. We consider the model where there is one designated master node and all other nodes are anonymous and have constant space. Recently, Lee et al. obtained such an algorithm for determining the size of a unidirectional ring. We provide a new algorithm that converges much quicker. This algorithm exploits a token-circulation idea due to Afek and Brown. Disregarding the time for stabilization, our algorithm computes the size of the ring at the master node in \( O(n \log n) \) time compared to \( O(n^3) \) steps used in the algorithm by Lee et al. using the same computing paradigm. It seems likely that one should be able to obtain master–slave algorithms for other problems in networks.

1 Introduction

Self-stabilizing algorithms represent an extension of distributed algorithms in which nodes of the network have neither coordination, synchronization, nor initialization. Thus self-stabilization handles an unlimited number of transient faults. Each node participates in the distributed algorithm based on local knowledge: its own state and the states of its immediate neighbors. The objective is to achieve some global objective—a predicate defined on the states of all the nodes in the network—based on local actions where individual nodes have no global knowledge about the network. See [1, 2] for a general overview of the paradigm of self-stabilization and its requirements.

Recently, Lee, Tzeng, and Huang [3] offered a new model for self-stabilization. They considered the problem of determining the number \( n \) of nodes in a ring using constant space. Well, actually this is impossible, since even writing down the value \( n \) requires \( O(\log n) \) space. So, to obtain a “space-efficient” algorithm, they designated one node that would determine the answer, but required all other nodes to use constant space.

We call their algorithm a master–slave algorithm. The network has a unique master node, and all other nodes are anonymous; thus the algorithms are semi-uniform. Anonymous uniform algorithms are theoretically more appealing, but the class of problems that can be solved in that model is rather small due to the fact that to establish many global primitives by local computation one needs be able to break symmetry between competing nodes. The class of semi-uniform algorithms, on the other hand, provides more computation power and hence can solve a larger class of problem. Moreover, these semi-uniform algorithms are closer to model sen-
sor or ad hoc networks where the sensor nodes are extremely resource challenged and a base station or a cluster head needs to determine some information. Recent efforts focus on resource-constrained devices, with an emphasis on in situ sensor nodes. TinyOS [4, 5, 6] was among the first operating systems so tailored and remains prominent. This master-slave model then would be useful for such wireless networks using cluster architecture [7, 8, 9].

In this paper, we provide a more efficient master–slave algorithm for the problem considered by Lee et al. This algorithm is more efficient in that it reduces the time for convergence. It is built on a token circulation algorithm of Afek and Brown [10]. The token circulation protocol uses three states at each slave node and the randomized version stabilizes in expected $O(n \log n)$ steps. Once the system stabilizes with one token, we then propose a polling algorithm to compute the size of the ring at the master node. This (deterministic) algorithm stabilizes in $O(n \log n)$ steps. We observe that the existing ring-size determination algorithm by Lee et al. [3] (the only one in the same computing model) stabilizes in $O(n^4)$ steps; our algorithm utilizes the same amount of storage at the master node and the slave nodes; the master node needs $O(\log n)$ space to compute the size of the ring while the slave nodes need constant space.

## 2 Terminology and Model

In a self-stabilizing algorithm, every node has a set of local variables whose contents specify the local state of the node. The state of the entire system, called the global state, is the union of the local states of all the nodes in the system. Each node has only a partial view of the global state (and this view depends on the connectivity of the system). Furthermore, there is no synchronization, not even a common starting point. Yet, the system arrives at a desirable global final state (legitimate state).

A self-stabilizing algorithm is specified as a collection of (production) rules for each node. Each rule has a trigger condition and an action. The trigger condition is a boolean predicate on the state of the node and the state of its neighbors; the action or move specifies a change in the state of the node’s variables. A node is said to be privileged at a particular time if the trigger condition on one or more of its rules is satisfied.

In order to analyze the correctness of the algorithm and its time complexity under a worst-case scenario, a self-stabilizing algorithm is assumed to face an adversarial daemon. The daemon plays the role of both scheduler and adversary. In the literature, there are several daemons. We focus here on a central daemon, also known as a serial daemon. A central daemon picks a single arbitrary privileged node to move (execute the rule’s action) at each step. In contrast, the distributed daemon taps a nonempty subset of the privileged nodes to move at each step. We show that the algorithms work under both daemons.

We assume that we have a unidirectional ring network, with the master node labeled 0 and the other nodes in order labeled 1 through $n-1$. These labels are used only for reference purposes; the master node is unique, it knows it’s the master and the other nodes are anonymous. Any node $i$ has a predecessor $i-1$ and the master node 0 has its predecessor $n-1$. As is customary in self-stabilization, we work in the shared-variable or state-reading model in which a node can directly read its predecessor’s variables.

The algorithms for token circulation and for ring-size computation do not terminate. Rather, they are guaranteed to eventually reach and remain in a state where there is a single token in circulation, or the master node knows the size of the ring, respectively. This is sometimes referred to as the algorithm not being silent.

## 3 Master–Slave Token Circulation in Rings

In a celebrated and elegant paper, Gouda and Haddix [11] provided a self-stabilizing token-circulating algorithm that uses three bits for each node, that is, eight
states. However, if one looks for a master–slave algorithm, then one can use just three states at each slave node. This algorithm is a direct adaption of an earlier algorithm of Afek and Brown [12] and is essentially given by them in [10]. The model is a tiny bit different in that they assume some initialization of the state of the nodes, and have FIFO queues on the links of the network, but the adaptation to this model is trivial and the analysis is somewhat similar. We call the result the AB algorithm.

In the AB algorithm, each node has a token variable \( t_i \) that can be in one of three states, say \( A, B, \) or \( C \). The rule for any slave node \( i \) is trivial: If \( t_i \neq t_{i-1} \), then \( t_i = t_{i-1} \). When node \( i \) changes its \( t \)-value, we say that it has received the token, and any desired actions are performed then.

The rule for the master node is based on the opposite idea: it becomes privileged when its value is the same as its predecessor, that is, if \( t_0 = t_{n-1} \). When privileged and chosen by the daemon, the master node changes its \( t \)-state. The interesting question is the rule for how the master node changes its token state. We call this the Update rule, and defer discussion about this until we have established some general properties of the algorithm. The AB algorithm is given in Figure 1.

**Figure 1. AB Algorithm for Token Circulation**

The desired goal behavior is that the master node changes its \( t \)-value, this value circulates the entire ring, and then the master node changes its value again. Indeed, we have a legal final configuration if and only if the number of privileged nodes is 1.

**Definition 1** Any global system state is **legal** or **stable** if and only if there is exactly one token in the ring, i.e., the number of privileged nodes (the master or a slave) is exactly one.

We note immediately that the algorithm remains live, and that over time the number of privileged nodes can only decrease:

**Lemma 1**
(a) The AB algorithm cannot terminate.
(b) The number of privileged nodes cannot increase.

**Proof:**
(a) If every slave node is not privileged, then this means that \( t_0 = t_1 = t_2 = \ldots = t_{n-1} \). Thus the master node is privileged.

(b) It is immediate that the number of privileged nodes can never go up—in any unidirectional ring algorithm, as a node moves, it becomes unprivileged and only its successor can be made privileged. □

Thus we only need argue that eventually the number of privileged nodes decreases. To get stuck in an infinite loop, what enters the master node must continually be the same as what leaves it. Thus, as Afek and Brown show, for convergence it is sufficient to choose a sequence for the \( t_0 \) such that the sequence is aperiodic:

**Lemma 2** [12] If the AB algorithm does not converge to a legal final configuration, then the sequence of distinct values of \( t_0 \) is periodic.

3.1 The Update Rule

We now turn to discussion of the Update rule. This amounts to discussion of ways of generation of aperiodic sequences. One complex idea is to use a square-free word [13] (such as that given by Thue [14]), which has the property that no subword is repeated, and in particular, is not periodic.

But a simple idea is that the master node maintains two counters: \( r \) and \( s \) (for round and step). The Update rule is given as:

If we were allowed to initialize the master node (to \( t = C \) and \( s = r = 0 \)), then this rule would generate

Deterministic Update:

\[
\begin{align*}
\text{if } (t &= A) \\
    t &= B; \\
\text{else if } (t &= A \text{ and } s < r) \\
    t &= B; \text{ } s++; \\
\text{else if } (t &= B \text{ and } s = r) \\
    t &= C; \\
\text{else} \\
    t &= A; \text{ } s = 0; \text{ } r++
\end{align*}
\]

Figure 2. Deterministic Update

This sequence is clearly aperiodic, as noted in [10]. However, this algorithm may require unbounded memory at the master node. Moreover, because the master node is not initialized, it can start in a state where \( s = 0 \) and \( r \) is huge. It will then produce alternating \( A, B \) for a very long time, and the daemon can avoid stabilization with such a sequence. Thus there is no upper bound on the stabilization time of the AB algorithm for the above deterministic update.

This situation can be avoided if one knows a bound \( B \) on the size of the ring (or possibly even if this algorithm is used in the midst of an algorithm that computes the size \( B \) of the ring). For, in this case, one can wrap \( r \) around to zero when it reaches \( B + 1 \) (a period longer than \( B \) is irrelevant). Thus one uses \( O(\log B) \) bits at the master node. We note that this is probably best possible.

However, one simple practical way to implement the AB algorithm is to add randomness, as observed in [10]. Specifically, allow the master node to choose the next \( t_0 \) uniformly at random from the two other values. It follows that if there are multiple tokens in circulation, what the master node receives from node \( n - 1 \) is not what the master just sent. Thus there is a 50-50 chance that the next token will match, and thus disappear.

Randomized Update:

Let \( t \in \{A, B, C\} \rightarrow t \) uniformly at random.

Figure 3. Randomized Update

As in [3], we define a round to be the time taken by a token to circulate around the ring; that is, a round is equivalent to \( n \) steps.

**Theorem 1** Under Randomized Update, the expected time to convergence of one circulating token is \( O(n \log n) \) steps, regardless of the behavior of the daemon.

**Proof:** We give just a sketch of the proof. The goal of the daemon is to maintain multiple tokens in circulation. Most moves simply advance a token from one node to the next without changing the set of tokens. The moves that can potentially destroy a token are (a) tapping the master node; or (b) advancing one token to overwrite the next (for example, if three consecutive slave nodes have \( A, B, C \) and the daemon taps the middle node then the \( B \)-token dies).

Consider a situation where the daemon performs (a). Say the first token in circulation is \( A \) and the daemon and its predecessor are in \( B \). Then there is a 50% chance that the token generated by the daemon is the same as that, and hence the token dies. Thus, if the daemon never does (b), then the expected number of tokens generated by the master node before we get to one token is at most \( 2n \). Thus, approximately, if we start with \( n \) tokens, after all have passed through the master we should expect to have about \( n/2 \) tokens, and so on, and so after \( O(\log n) \) rounds we would have only one token.

Now, to provide a full proof, we need to make the previous sentence more mathematically accurate. But more importantly, we need to deal with the fact that the daemon can use (b) to destroy specific tokens. For example, this behavior is especially noticeable if the sequence of tokens is the deterministic one given above. There if the daemon destroys every \( C \) that is generated and one associated \( B \), it can force \( O(n) \) rounds to convergence (even if we are allowed to initialize the master node). So one needs to show that option (b) does not change things significantly. We omit the details. \( \square \)

**Remark 1** The AB algorithm handles a distributed
daemon too. Indeed, if the distributed daemon does not choose all the nodes simultaneously, then in any unidirectional ring algorithm, this is equivalent to a sequence of individual moves (move the frontmost node first). So the only issue with considering a distributed daemon is what happens if the daemon selects all nodes simultaneously. This can, of course, only happen if every node is privileged. And we have just argued that that state cannot continue indefinitely.

4 Better Size Computation on Rings

In [3], the authors proposed a self-stabilizing algorithm for measuring the size of a unidirectional ring network. They used the idea of circulating a low-level token and a high-level token. Whenever the low-level token hits the high-level token, the high-level token moves forward one node, but the low-level token continues past. Thus the master node counts how many times it sees the low-level token in between seeing the high-level token.

That algorithm takes time proportional to $O(n^3)$ rounds or circulations of the ring, ignoring stabilization period. The time taken, starting from an arbitrary system state to stabilize to a state of a single high-level token and a single low-level token, is $O(n^3)$ rounds. That protocol requires that each node has five boolean variables and the root node (master node) has two additional integer variables (to eventually accumulate the size of the ring).

4.1 Informal Description of Our Approach

We use a different idea to compute the size of the ring. This is based on the standard algorithm for counting a set of cells in a 1-tape Turing Machine: compute the count one bit at a time. The master node determines the ring-size one bit per round (in a right-to-left fashion) such that $O(\log n)$ circulations of the token suffice, thereby significantly reducing the execution time.

To explain the approach, we ignore self-stabilization for the time being. The concept of the distributed algorithm is as follows. Every slave node has an Alive bit. The master node starts by sending round a Reset token that sets every node’s Alive bit to true. After that, the master sends out a Counter token with a single bit. The first round, this Counter token determines the parity of the number of nodes, since every node toggles the Counter bit.

Furthermore, the master node sends the token out with Counter bit clear. Every node toggles the bit; but those nodes that set the bit also clear their Alive bit. The second time the Counter token circulates, it does the same thing, except that nodes that are Dead simply pass on the Counter token unchanged. In this way, the master node receives the parity of $\lfloor n/2 \rfloor$. And so on.

In $\log n$ rounds, the master node can calculate the total number of nodes. In order to get the master to know that the process is complete, one adds another “Pristine” bit to the token; this state is cleared by the first node to toggle the Counter bit. If the master node gets the Pristine state back, then it knows all nodes are dead and the algorithm is complete.

4.2 Protocol RING

To turn the above description into a self-stabilizing algorithm, we simply use the AB algorithm for circulating a token, as developed in the previous section. In the RING algorithm, each node $i$ maintains following data structure:

- A 3-state token variable $t_i$ that can have any of the three values $A$, $B$, or $C$ (this can be implemented by using only 2 bits).

- A 4-state status variable $status_i$; that can have any of four values Pristine, Reset, Zero, or One (this can be implemented by using only 2 bits).

- Each slave node $i$ ($i > 0$) has a one-bit boolean flag $live_i$. We say that a slave node $i$ is alive if $live_i$ is true; otherwise, it is dead.
• The unique master node \(i = 0\) has two integer variables \(\text{count}\) and \(\text{pos}\) to store the size of the ring.

In the algorithm, the unique master node is privileged (i.e., has a token) if \(t_0 = t_{n-1}\), and any other (slave) node \(i > 0\) is privileged (i.e., has a token) if \(t_i \neq t_{i-1}\). Token circulation is done using the previous self-stabilizing AB algorithm, using the \(t\)-tokens.

We view the status variable as if the node has a status token—a privileged node receives a status token from its predecessor, adjusts its own status variable according to the rules, and thus sends the status token to the next node.

The master node sends out a \(R\) status token only when it receives a \(P\) status token from its predecessor (node \(n - 1\)). Otherwise, the master node always sends out a \(P\) status token. If a dead slave node receives a non-\(R\) status token, it relays the status token unchanged to its successor and stays dead. If an alive slave node receives a non-\(R\) status token, it moves based on the received token. If it receives a \(P\) or \(Z\) status token, then it changes it to an \(O\) status token and the node becomes dead; if it receives an \(O\) status token, then it changes it to a \(Z\) status token and the node stays alive.

The complete details for the RING protocol are shown in Figure 4.

4.3 Analysis

A configuration as legal if there is exactly one \(t\)-token in the system (either \(A\), \(B\), or \(C\) type), i.e., exactly one node (either slave or master) is privileged. When the privileged node moves, the system transitions to a legal configuration where the node \(i + 1\) is privileged. Thus, at each step, the single system token advances to the next node in the unidirectional ring.

By Theorem 1, the global system state will reach a legal configuration in expected \(O(n \log n)\) steps starting from an arbitrary system state.

Lemma 3 When the master node is privileged and sends out an \(R\) status token, all slave nodes becomes alive in the next round whereupon the master node becomes privileged again.

Proof: When a privileged slave node receives a \(R\) status token, it sets its live bit to be true and sends an \(R\) status token to the next node (by the first clause in the rules for slaves). The process continues all the way to the master node.

Lemma 4 In a round, if the master does not send an \(R\) status token, then (1) the number of alive slaves cannot increase; and (2) moreover, at least half of the alive slave nodes at the beginning of the round will be dead by the end of the round.

Proof: We saw earlier that a dead slave node, when privileged, remains dead and relays the status token received from its predecessor to its successor. Thus (1) is true.

For (2), we recall that an alive slave node, when privileged, will become dead after sending out an \(O\) status token on receiving a \(P\) or \(Z\) status token. Thus an alive node can stay alive in a round only if it receives an \(O\) status token from its predecessor. But then, it sends out a \(Z\) status token to its successor and hence the next alive slave will become dead in that round. Thus, alternate alive nodes in the ring become dead by the end of the round.

Lemma 5 Starting with any good configuration, all alive nodes will be dead by at most \(\lceil \log_2 n \rceil\) rounds or in \(O(n \log n)\) steps.

Proof: In any configuration, the number of alive slave nodes is at most \(n - 1\). Thus this lemma follows from Lemma 4.

Lemma 6 When all slave nodes are dead, the master node will receive a \(P\) status token in at most one round.
Rule for the Master Node ($i = 0$):
if $t_0 = t_{n-1}$ then
\[
\begin{align*}
&\text{Set } t_0 = \text{choose by Update from } \{A, B, C\} - \{t_{n-1}\}; \\
&\text{if } status_{n-1} = R \text{ then} \\
&\quad\text{\{ } status_0 = P; count = 0; pos = 0; \text{\} } \\
&\text{else if } status_{n-1} = P \text{ then} \\
&\quad\text{\{ } status_0 = R; count += 1; \text{\} } \\
&\text{else if } status_{n-1} = Z \text{ then} \\
&\quad\text{\{ } status_0 = P; pos += 1; \text{\} } \\
&\text{else if } status_{n-1} = O \text{ then} \\
&\quad\text{\{ } status_0 = P; count += 2^{pos}; pos += 1; \text{\} }
\end{align*}
\]

Rule for a Slave Node ($i > 0$):
if $t_i \neq t_{i-1}$ then
\[
\begin{align*}
&\text{Set } t_i = t_{i-1}; \\
&\text{if } status_{i-1} = R \text{ then} \\
&\quad\text{\{ } status_i = R; live_i = true; \text{\} } \\
&\text{else if } status_{i-1} = P \text{ then} \\
&\quad\text{\{ if live_i then } status_i = O; live_i = false; \text{\} else } status_i = P; \text{ \} } \\
&\text{else if } status_{i-1} = Z \text{ then} \\
&\quad\text{\{ if live_i then } status_i = O; live_i = false; \text{\} else } status_i = Z; \text{ \} } \\
&\text{else if } status_{i-1} = O \text{ then} \\
&\quad\text{\{ if live_i then } status_i = Z; \text{ else } status_i = O; \text{ \} }
\end{align*}
\]

Figure 4. Master–Slave Algorithm RING to compute the size of an Unidirectional Ring

Proof: The master node becomes privileged once each round. If it does not receive a $P$ status token, then it will relay a $P$ status token. This status token will circulate around the ring since dead nodes just pass the token, and so will reach the predecessor of the master node. □

Putting all this together, we get the following:

Theorem 2 Starting from an arbitrary legal configuration, the integer variable count at the master node will contain the size $n$ of the ring in at most $2(\lceil \log_2 n \rceil + 1)$ rounds (O($n \log n$) steps).

Proof: By Lemmas 5 and 6, the system will reach a special legal configuration where all slave nodes are dead and the master node is privileged with an $R$ status token in at most $\lceil \log_2 n \rceil + 1$ rounds.

At this point, the master node sends out a $R$ status token. This makes in one round each slave node $i > 0$ alive with its $status_i$ set to $R$. On receiving the $R$ status token, the privileged master node, in the next round, sends out a $P$ status token. Slave node $i = 1$ will send out an $O$ status token and will become dead; as the status token circulates the ring, alternate slave nodes will become dead. At the end of the round, the privileged master node will receive a status token $O$ or $Z$ depending on whether the number of slave nodes is odd or even. The master adjusts the counter variable (incrementing it by $2^{pos}$ on an $O$ and doing nothing on a $Z$) and incrementing position by 1 (the variable $pos$ keeps track of the number of rounds), and starts a new round in a similar way. After $\lceil \log_2 n \rceil$ rounds, the master node gets back the status token $P$; it now knows that the counter contains the correct number of slave nodes and so, adds 1 to it to account for itself. □
5 Conclusion

We have shown how to produce a faster algorithm for determining the size of a ring using constant space at all nodes except the one needing to know the size. Disregarding the time for stabilization, our algorithm computes the size of the ring at the master node in \(O(n \log n)\) time compared to \(O(n^3)\) steps taken by the algorithm in [3] using the same computing paradigm. It seems likely that one should be able to obtain master–slave algorithms for other problems in networks.

References


