RESOURCE BOUNDS FOR SELF-STABILIZING MESSAGE DRIVEN PROTOCOLS

SHLOMI DOLEV*, AMOS ISRAELI†, AND SHLOMO MORAN‡

Abstract. Self-stabilizing message driven protocols are defined and discussed. The class *weak-exclusion* that contains many natural tasks such as *ℓ-exclusion* and *token-passing* is defined, and it is shown that in any execution of any self-stabilizing protocol for a task in this class, the configuration size must grow at least in a logarithmic rate. This last lower bound is valid even if the system is supported by a time-out mechanism that prevents communication deadlocks. Then we present three self-stabilizing message driven protocols for token-passing. The rate of growth of configuration size for all three protocols matches the aforementioned lower bound. Our protocols are presented for two processor systems but can be easily adapted to rings of arbitrary size. Our results have an interesting interpretation in terms of automata theory.

Key words. self-stabilization, message passing, token-passing, shared-memory

AMS subject classifications. 68M10, 68M15, 68Q10, 68Q20

1. Introduction. A distributed system is a set of state machines, called *processors*, which communicate either by *shared variables* or by *message-passing*. In the first case, the system is a *shared memory* system, in the second case the system is a *message-passing* system. A distributed system is *self-stabilizing* if it can be started in any *possible* global state. Once started, the system regains its consistency by itself, without any kind of an outside intervention. The self-stabilization property is very useful for systems in which processors may crash and then recover spontaneously in an arbitrary state. When the intermediate period in between one recovery and the next crash is long enough, the system-stabilizes. Self-stabilizing systems were defined and discussed first in the fundamental paper of Dijkstra, [7]. The work of [7] as well as most of the following work on self-stabilizing systems assume the communication model of shared variables. Among these papers are [17], [22], [8], [20], [2], [6], [4], [14], [15], [9] and [11].

In the study of fault tolerant message-passing systems, it is customarily assumed that messages might be corrupted over links, hence, processors may enter arbitrary states and link contents may be arbitrary. Self-stabilizing protocols treat these problems naturally, since they are designed to recover from inconsistent global-states. Surprisingly, there are very few papers which address self-stabilizing, message-passing systems. The earliest research in this model was done by Gouda and Multari in [21, 13]. In that work, they have developed a self-stabilizing sliding window protocol and two-way handshake that use unbounded counters. They proved that any self-stabilizing message passing protocol must use time-outs and have infinite number of safe states. Following [13], two additional works dealt with self-stabilizing protocols in this model: The work of Katz and Perry, [16], presents a general tool for extending an arbitrary message-passing protocol to a self-stabilizing protocol. The work of Afek and Brown, [1], presents a self-stabilizing version of the well-known alternating-bit protocol, (see e.g. [5]).

* Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel, dolev@cs.bgu.ac.il.

† Intel, Haifa 31015, Israel aisraeli@iil.intel.com. Partially supported by NWO through NFI Project ALADDIN under Contract number NF 62-376.

‡ Department of Computer Science, Technion, Haifa 32000, Israel, moran@cs.technion.ac.il.

In this work we research complexity issues related to self-stabilizing, message-passing systems; to do that we define a *configuration* of any message-passing system as a list of the states of the processors and of the messages which are in transit on each link. The *size* of a configuration of a message-passing system is the number of bits required to encode the configuration entirely. A protocol for a message-passing system is *message-driven* if any action of the processors is initiated by receiving a message. In the work of Gouda and Multari, [13], it is proven that any message-driven protocol has a possible configuration in which all processors are waiting for messages but there are no messages on any link. This unwanted situation is called *communication deadlock*. A self-stabilizing system should stabilize when started from any possible initial configuration, including a configuration with communication deadlock. This implies that a non-trivial, completely asynchronous, self-stabilizing system cannot be message-driven. This problem can be dealt with in at least two methods: Gouda and Multari, in [13], proposed the use of a time-out mechanism which preserves the message driven structure of the protocol at the expense of compromising the complete asynchronisity. On the other hand, Katz and Perry, in [16], have chosen to give up the message-driven structure and present protocols for which at any configuration there is at least one processor whose next operation is sending a message. Thus, there is an execution in which in every atomic step a message is sent, and no message is ever received. In this execution the size of the configurations grows *linearly*.

In this work we define and study the class of *self-stabilizing, message-driven* protocols. By the argument of [13], there exists no self-stabilizing, message-driven protocol which is completely asynchronous. Since we look for protocols whose configuration size does not grow in linear rate we resort to slightly limited assumptions of asynchronous behavior. For lower bounds we assume an abstract time-out device which detects communication deadlocks and initiates the system upon their occurrence. Consequently, the lower bounds we present take into account only executions in which no communication deadlock occurs. Our upper bounds assume that in every initial configuration there is at least one message on some link. This assumption is much weaker than the assumption on a general time-out mechanism.

A specific task which we study in details is *token-passing*. Informally, the *token-passing* task is to pass a single token fairly among the system's processors. Usually it is assumed that in the system's predefined initial configuration there exists a single token. In self-stabilizing system in which there is no predefined initial configuration, each execution should reach a configuration in which exactly one token is present in the entire system. Token-passing is a very basic task in fault tolerant systems, among other works it was studied in [12] for some fault tolerant message-passing systems and in [14], for self-stabilizing, shared memory systems. The token-passing task can be looked at as a special case of mutual-exclusion since possession of the single token can be interpreted as a permission to enter the critical section.

In the first part of the presentation we prove a lower bound on the configuration size for protocols for a large class of tasks called *weak-exclusion*. The weak-exclusion class contains all non-trivial tasks which require continuous changes in the system's configuration; in particular this class includes both ℓ*-exclusion* and *token-passing*. We show that the configuration size of any self-stabilizing protocol which realizes any weak-exclusion task is at least logarithmic in the number of steps executed by the protocol. The lower bound holds for message-driven protocols for any week-exclusion task, including protocols for systems equipped with time-out mechanism. This result should be compared with a result of [13] where it is shown that any message-driven,

self-stabilizing protocol (not necessarily for week-exclusion task) must have infinitely many safe system configurations, but not that each specific execution must contain infinitely many distinct configurations, as implied by our results. Our lower bound does not specify which *part* of the system grows, is it the size of the memory used by the state machines, the size of messages stored on the links, the number of messages stored on the links or all of these together?

We then present three self-stabilizing, message-driven protocols for token-passing. The communication deadlock problem is avoided by the assumption that at least a single message is present on some communication link. Using this assumption, we present three token-passing protocols, for two processors each. The rate of growth of configuration size for all three protocols matches the aforementioned lower bound. All protocols are presented for systems with two processors but can be easily adapted to work on rings of arbitrary size *without increasing their asymptotic complexity*. This is done by considering the ring as a single virtual link.

In the first protocol both processors memory and messages size grow unboundedly with time, this protocol uses ideas similar to the ideas of the sliding window protocol of [13]. The second protocol is an improvement on the first protocol in which the size of the memory of the processors grows (in logarithmic rate) while the size of the link content is bounded. The second protocol is an improvement of the deterministic alternating bit protocol of [1]. The third protocol is a self-stabilizing token-passing protocol in which processors are *deterministic* finite state machines and messages are of fixed size. The only growing part of the system is the number of messages on the links; the rate of growth matches the lower bound mentioned above.

Our results can be described also in terms of automata theory, as follows: Let Σ be an alphabet. Define a *queue machine* Q to be a finite state machine which is equipped with a queue, which initially contains an arbitrary non empty word from Σ^+. Initially Q is in an arbitrary state, and in each step it performs the following: (a) reads and deletes a letter from the head of the queue, (b) adds one or more letters from Σ to the tail of the queue, and (c) moves to a new state. The computational power of a queue machine is severely limited by the fact that its input alphabet and its work alphabet are *identical*. In particular a queue machine cannot perform simple tasks like computing the length of the input word, or even deciding whether the input word contains a specific letter.

Assume that the alphabet contains a specified subset τ of *token letters*. A queue machine is a *token-controller* if, starting with a nonempty queue of arbitrary content, eventually the queue contains exactly one occurrence of a letter from τ forever. Our lower bound result implies that if a token-controller exists, then in every computation the size of the queue must grow at least logarithmic in the number of moves of the machine. Our third protocol implies that a token-controller whose configuration size growth matches the lower bound exists. In view of the fact that a queue machine *cannot* compute any estimation of the number of occurrences of letters from τ in the input word, this latter result appears to be somewhat counter intuitive.

2. Self-Stabilizing Message-Driven Systems.

2.1. Asynchronous Message-Driven Systems. An *asynchronous, distributed, message-passing system* contains n processors where each processor is a state machine. Processors communicate using message-passing along *links*. An edge $e = (i, j)$ of G stands for two directed links, one from P_i to P_j and the other from P_j to P_i. A message sent from P_i to P_j can be delayed for an unbounded amount of time on the

connecting link. Messages which did not reach their destination yet, are stored on the link and transferred in FIFO (First In First Out) order.

A processor is uniquely defined by the set of its *atomic steps*. Whenever a processor is active it executes one of its atomic steps. In a *message-driven* protocol an atomic step of any processor P begins with a **receive** operation in which P receives a message from one of its incoming links. The atomic step ends with zero or more **send** operations in which P sends messages along some of its outgoing links. An atomic step a of P_i is defined by $a = (i, s_{i_1}, (e, msg), (e_1, msg_1), (e_2, msg_2) \cdots (e_\ell, msg_\ell), s_{i_2})$ meaning that: P_i is in state s_{i_1}, e is the link through which P_i receives the message msg, $e_1, e_2, \cdots, e_\ell$ are the outgoing links along which P_i sends $msg_1, msg_2, \cdots, msg_\ell$, respectively and s_{i_2} is the state of P_i following the execution of this atomic step.

Let n and m be the number of processors and links respectively in the system. For $1 \leq i \leq n$ denote by S_i the set of states of P_i. A *configuration* of the system is a vector of states of all processors together with m lists, a list for every link, of messages stored on that link. A configuration is denoted by c=$(s_1 \times s_2 \times \cdots s_n \times M_{e_1} \times M_{e_2} \cdots \times M_{e_j} ... \times M_{e_m})$ where $s_i \in S_i$, $1 \leq i \leq n$, and M_{e_j} is a list of the messages stored on e_j, for $1 \leq j \leq m$. Let c be a configuration as above, and let $a = (i, s_{i_1}, (e, msg), (e_1, msg_1), (e_2, msg_2) \cdots (e_\ell, msg_\ell), s_{i_2})$ be an atomic step. a is *applicable* to (P_i in) c, if P_i is in state s_{i_1} in c and msg is the first message stored on e in c.

Application of a to c yields the *result* configuration c'. We denote this fact by $c \xrightarrow{a} c'$. A sequence of atomic steps, $A = (a_1, a_2, \cdots)$, is applicable to configuration c_0, if the first atomic step in the sequence, a_1, is applicable to c_0, the second atomic step is applicable to c_1 where $c_0 \xrightarrow{a_1} c_1$, and so on. An *execution*, $E = (c_0, a_1, c_1, a_2, \cdots)$ is a (finite or infinite) sequence which starts with some arbitrary configuration c_0 and for every $i > 0$, $c_{i-1} \xrightarrow{a_i} c_i$, that is: the sequence of atomic steps $A = (a_1, a_2, \cdots)$ is applicable to c_0. Note: Since we deal with self-stabilizing systems we do not assume any particular initial configuration, every configuration is a valid initial configuration. Execution E is *fair* if every atomic step that is applicable infinitely often is executed infinitely often.

Each execution E defines a partial order on the atomic steps of E by the relation *happened before* of Lamport in [18]:

1. If a_i and a_j are atomic steps executed by the same processor in E and a_i appears before a_j in E, then a_i *happened before* a_j.
2. If during a_i the message msg is sent and during a_j the same message msg is received, then a_i *happened before* a_j.
3. If a_i *happened before* a_j and a_j *happened before* a_k then a_i *happened before* a_k.

We also adopt the definition of *concurrent* atomic steps from [18]: atomic steps $a_1 \cdots a_k$ are said to be *concurrent* in an execution E if for $1 \leq i < j \leq k$, a_i does not *happen before* a_j and a_j does not *happen before* a_i in E. The following proposition gives a sufficient condition for a set of steps to be concurrent in some execution:

PROPOSITION 2.1. *Let $P_{i_1}, \cdots, P_{i_k}$ be k distinct processors and let $\{a_1, \cdots, a_k\}$ be a set of atomic steps where a_j is applicable to P_{i_j}, $1 \leq j \leq k$ in some configuration c. Then there exists an execution in which the atomic steps $a_1 \cdots a_k$ are concurrent.*

Proof. Observe that once step a is applicable to processor P in configuration c, step a remains applicable to P in all subsequent configurations. The execution E is defined as the execution that starts from c, in which processors $P_{i_1}, \cdots, P_{i_k}$ are activated one after the other, and each processor P_{i_j} executes a_j. The proof follows

since the processors are distinct and since in E, no message that was sent during a_j, $1 \leq j \leq k$, is received before a_k is executed. Note that the proposition holds for any system in which once some step is applicable it remains applicable as long as it is not executed. □

An asynchronous protocol, PR, is defined by a set of n processors. By the above definitions, an asynchronous protocol defines a set of executions that satisfy the following:

1. Let $E = (c_0, a_1, c_1, a_2, \cdots)$ be an arbitrary execution of PR. Then every prefix of E is also an execution of PR.
2. Let $E = (c_0, a_1, c_1, a_2, \cdots, a_r, c_r)$ be arbitrary finite execution of PR. Then for every atomic step a and configuration c, satisfying $c_r \xrightarrow{a} c$, PR has an execution $E \circ (a, c)$[1].

2.2. Self-Stabilizing Message-Driven Protocols. A self-stabilizing system demonstrates a *legitimate behavior* some time after it is started from an arbitrary configuration. A natural way to specify a behavior in an abstract way is by a set of sequences of configurations. We define *tasks* as sets of *legitimate-sequences*. The semantics of any specific task is expressed by requirements on its sequences. Intuitively each legitimate sequence can be thought of as an execution of a protocol but we do not require it formally. For instance, the mutual-exclusion task is defined as the set of sequences of configurations which satisfy: Each processor has a subset of its states called *critical section*; in each configuration, at most one processor is in its critical section, and every processor is in its critical section in infinitely many configurations. To formally define a task T, one should specify for each possible system ST, a set of *legitimate sequences* for ST. The *task* T is defined as the union of the legitimate sequence set over all possible systems. A configuration c of a system is *safe* with respect to a task T and a protocol PR if any fair execution of PR starting from c belongs to T.

In proving lower bound results on self-stabilizing message-driven protocols, we assume that the system can recover from a *communication deadlock* (called deadlock from now on). In other words: When we prove our lower bounds, we assume only that the protocol stabilizes in executions in which no deadlock occurs. For this purpose, we distinguish between two types of deadlocks: *global* and *local*. A configuration c is a global deadlock configuration if no atomic step is applicable to c. Our first lower bound holds for asynchronous systems that can recover from global deadlocks by applying a *global time-out* mechanism. This abstract mechanism initiates a system in a global deadlock configuration to a default initial configuration, after which no deadlock occurs. Below we bring the requirement for self-stabilizing systems equipped with a global time-out mechanism. In this definition the system is required to reach a safe configuration in every infinite fair execution. Note that by our definition an infinite fair execution does not have a deadlock configuration.

[Self-Stabilization] - *assuming global time-out mechanism*

Let PR and LE be a message driven protocol and set of legitimate sequences, respectively. Protocol PR is self-stabilizing relative to LE, if for every c, there is an execution of PR that starts with c and every such infinite fair execution reaches a safe configuration with respect to LE and PR.

Later on, we prove a lower bound that holds for systems immuned from a stronger type of communication deadlock called local deadlock. Processor P is in a *local*

[1] For sequences S_1 and S_2, $S_1 \circ S_2$ denotes the concatenation of S_1 and S_2.

deadlock during execution E, if P is activated (i.e. executes an atomic step) only finitely many times during E. The second lower bound holds for systems equipped with an abstract *local* time-out mechanism which prevents such executions (e.g. by enabling each processor which is idle for a sufficiently long time to initiate the system to some default configuration after which no deadlock is possible). Note that a local time-out mechanism is strictly stronger than a global time-out mechanism.

[Self-Stabilization] - *assuming local time-out mechanism*

Let PR and LE be a message driven protocol and set of legitimate sequences, respectively. Protocol PR is sellf-stabilizing relative to LE, if for every c, there is an execution of PR that starts with c, and every such infinite fair execution, in which each processor is activated infinitely often, reaches a safe configuration with respect to LE and PR.

3. Lower Bound. In this section we prove a lower bound on the rate in which the configuration size grows along every execution of any protocol for a large class of tasks called *weak-exclusion*. This class contains all non-trivial tasks which require continuous changes in the system's configuration; in particular this class includes both *ℓ-exclusion* and *token-passing*. For an execution E, denote by $\mathcal{A}_i(E)$ the set of distinct atomic steps executed by P_i during E. A task belongs to the class *weak-exclusion* if its set of legitimate sequences, LE, satisfies:

[WE]- For any $E \in LE$ there exists a set of two or more atomic steps $B = \{a_{i_1}, \cdots, a_{i_k}\}$, $k \leq n$, where $a_j \in \mathcal{A}_{i_j}(E)$, such that the atomic steps in B are never concurrent during E.

We first consider self-stabilizing protocols for systems equipped with a global time-out mechanism. For these protocols we prove that in every execution (in which no communication deadlock occurs) all configurations are distinct. From this we conclude that the configuration size of every self-stabilizing protocol which realizes any weak-exclusion task is at least logarithmic in the number of steps executed by the protocol. Throughout the proof we assume that PR is a self-stabilizing, message-driven protocol for an arbitrary weak-exclusion task, in a system with a global time-out mechanism. At the end of this section, we present a slightly weaker lower bound for systems with a local time-out mechanism.

For any configuration c and any link e, denote by M_e^c the sequence of messages present on e in c. For any execution E, denote by $M_{e,s}^E$ ($M_{e,r}^E$) the sequence of messages sent (received) along e during E.

PROPOSITION 3.1. *For every execution $E = (c_0, a_1, \cdots, a_r, c_r)$ and for every link e, $M_e^{c_0} \circ M_{e,s}^E = M_{e,r}^E \circ M_e^{c_r}$.*

Proof. The left hand side of the equation contains the messages present on e in c_0, concatenated with the messages sent during E, through e. The right hand side of the equation contains the messages received during E through e, concatenated with the messages left on e in c_r. It is not hard to verify that both sides of the equation represent the same sequence of messages. □

An execution $E = (c_0, a_1, \cdots, c_{\ell-1}, a_\ell, c_\ell)$ whose result configuration c_ℓ is equal to its initial configuration c_0 is called a *circular* execution. A link e is *active* in a circular execution E if some messages are received (and hence, by the circularity of E, some messages are sent) along e in E. Repeating a circular execution E forever yields an infinite execution E^∞ which is not necessarily fair — The original execution may have an applicable step a which is never executed during E. The step a is applicable throughout E^∞ but it is never executed. To avoid this problem the original circular execution is changed by removing all messages from links that are not active

throughout E. The result execution, which is still called E is still circular and its infinite repetition E^∞ is a fair infinite execution. Observe that an execution in which a certain configuration appears more than once has a circular sub-execution, $\overline{E} = (c_i, a_{i+1}, \cdots, a_{i+\ell}, c_{i+\ell}) \equiv (\overline{c_0}, \overline{a_1}, \cdots, \overline{a_\ell}, \overline{c_\ell})$, where $c_i = c_{i+\ell} = \overline{c_0} = \overline{c_\ell}$. Thus, to show that in every execution of PR all the configurations are distinct, we assume that PR has a circular sub-execution $\overline{E}$ and reach a contradiction by showing that PR is not self-stabilizing.

Using $\overline{E}$, we now construct an initial configuration c_{init} by changing the list of messages in transit on the system's links. For each link e, the list of messages in transit on e, at c_{init}, is obtained by concatenating the list of messages in transit on e at $\overline{c_0}$ with the list of all messages sent on e during $\overline{E}$. Roughly speaking, the effect of this change is creating an additional "layer" of messages that helps to decouple each *send* from its counterpart *receive* and achieve an additional flexibility in the system which enables the proof of the lower bound: Formally, c_{init} is obtained from $\overline{c_0}$ as follows:

- The state of each processor in c_{init} is equal to its state in $\overline{c_0}$.
- For any active link in $\overline{E}$, $M_e^{c_{init}} = M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}}$ and for any non-active link in $\overline{E}$, $M_e^{c_{init}}$ is empty.

Let $\overline{A}(i)$ be the sequence of atomic steps executed by P_i during $\overline{E}$. Define $merge(\overline{A})$ to be the set of sequences obtained by all possible mergings of all sequences $\overline{A}(i)$, $1 \leq i \leq n$, while keeping the internal order in each $\overline{A}(i)$. Note that all the sequences in $merge(\overline{A})$ have the same finite length and contain the same atomic steps in different orders.

LEMMA 3.2. *Every $A \in merge(\overline{A})$ is applicable to c_{init}, and the resulting execution, $E_A = (c_{init}) \circ A$, is a circular execution of PR.*

Proof. Let A be an arbitrary sequence in $merge(\overline{A})$ and let P_i be an arbitrary processor of the system. Then we have: (i) The initial state of P_i in c_{init} is equal to its initial state in $\overline{c_0}$. (ii) In c_{init} all messages which P_i receives during $\overline{E}$ are stored on P_i's appropriate incoming links in the right order. (iii) The atomic steps of P_i appear in A in the same order they appear in $\overline{A}(i)$. (i) - (iii) above imply that the sequence A is applicable to c_{init}, and the application of A to c_{init} yields an execution, E_A, with result configuration, c_{res} whose state vector is equal to the state vector of c_{init} and in which for every active link $M_{e,s}^{E_A} = M_{e,s}^{\overline{E}}$ and $M_{e,r}^{E_A} = M_{e,r}^{\overline{E}}$.

To prove that the obtained execution is circular it remains to be shown that the content of every link in the result configuration, c_{res}, is equal to its content in c_{init} i.e. $M_e^{c_{init}} = M_e^{c_{res}}$. For any arbitrary link e it holds that:

1. $M_e^{c_{init}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{c_{res}}$ (by Proposition 3.1 and by the fact that $M_{e,s}^{E_A} = M_{e,s}^{\overline{E}}$ and $M_{e,r}^{E_A} = M_{e,r}^{\overline{E}}$).
2. $M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{\overline{c_0}}$ (by Proposition 3.1 and the circularity of $\overline{E}$).

Replacing $M_e^{c_{init}}$ in equation 1 with its explicit contents yields:

3. $M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{c_{res}}$.

Using equation 2 to replace $M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}}$ by $M_{e,r}^{\overline{E}} \circ M_e^{\overline{c_0}}$ in equation 3 gives:

4. $M_{e,r}^{\overline{E}} \circ M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{c_{res}}$.

Dropping $M_{e,r}^{\overline{E}}$ from the two sides of equation 4 yields the desired result: $M_e^{c_{init}} = M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} = M_e^{c_{res}}$, which proves the lemma. □

Define $blowup(\overline{E})$ to be the set of executions whose initial sate is c_{init} and whose sequence of atomic steps belongs to $merge(\overline{A})$. Notice that, for every circular execu-

tion $\overline{E}$ and for every execution $E \in blowup(\overline{E})$ it holds that $\mathcal{A}_i(\overline{E}) = \mathcal{A}_i(E)$.

LEMMA 3.3. *For any set of atomic steps $B = \{a_1, \cdots, a_k\}$, $k \leq n$, where $a_j \in \mathcal{A}_{i_j}(\overline{E})$, there is an execution $E \in blowup(\overline{E})$ that contains a configuration for which all the atomic steps in B are concurrent.*

Proof. For notational simplicity, assume that $k = n$ and that $B = \{a_1, a_2, \cdots, a_n\}$. Let $A \in merge(\overline{A})$ be the sequence constructed as follows: first take all the steps in $\overline{A}(1)$ that precede a_1, then take all the steps in $\overline{A}(2)$ that precede a_2,..., then take all the steps in $\overline{A}(n)$ that precede a_n. Applying the sequence constructed so far to c_{init} results in a configuration in which all the a_i's are applicable. This sequence is completed to a sequence A in $merge(\overline{A})$ by taking the remaining atomic steps in an arbitrary order, which keeps the internal order of each $\overline{A}_i$. □

LEMMA 3.4. *Let PR be a self-stabilizing, message-driven protocol for an arbitrary weak-exclusion task T, in a system with a global time-out mechanism. If PR has a circular execution $\overline{E}$ then PR has an infinite fair execution E^∞ none of whose configuration is safe for T.*

Proof. Let E be an arbitrary execution in $blowup(\overline{E})$. Define E^∞ to be the infinite execution obtained by repeating E forever). By the definition of $blowup(\overline{E})$, E^∞ is fair. So it remains to show that no configuration in E^∞ is safe.

Assume by way of contradiction that some configuration c_0 in E^∞ is safe. Now, we construct a finite circular execution E' whose sequence of atomic steps A' is obtained by concatenating sequences from $merge(\overline{A})$, that is $\mathcal{A}_i(E') = \mathcal{A}_i(\overline{E})$. Since PR is a protocol for some weak-exclusion task, E' should have some set of atomic steps $B = \{a_1, \cdots, a_k\}$, where $a_j \in \mathcal{A}_{i_j}$ that are never applicable for a single configuration c during E'. We reach a contradiction by refuting this statement for E': For this we choose some arbitrary enumeration $\mathcal{B} = B_1, \cdots, B_s$, of all the sets containing n atomic steps of n distinct processors. Execution E' is constructed by first continuing the computation from c_0 as in E until configuration c_{init} is reached. Then apply Lemma 3.3 to extend E' by s consecutive executions $E_1, \cdots, E_s$, where E_k, $1 \leq k \leq s$ contains a configuration in which all the steps in B_k are applicable and that ends with c_{init}. The proof follows. **Note:** Execution E' can be repeated forever to obtain an infinite execution which does not have any suffix in LE, thus, the protocol PR is not even pseudo self-stabilizing (see [3]). □

The proof for the lower bound is completed by the following theorem:

THEOREM 3.5. *Let PR be a self-stabilizing, message-driven protocol for an arbitrary weak-exclusion task, in a system with a global time-out mechanism. For every execution E of PR, all the configurations of E are distinct. Hence, for every $t > 0$, the size of at least one of the first t configurations in E is at least $\lceil \log_2(t) \rceil$.*

Proof. Assume by way of contradiction that there exists an execution E of PR in which not all the configurations are distinct, then E contains a circular sub-execution, $\overline{E}$. By Lemma 3.4, there exists an infinite execution E' of PR, which is obtained by an infinite repetition of some execution from $blowup(\overline{E})$, and which never reaches a safe configuration, a contradiction. □

For proving a similar lower bound to systems with a *local* time-out mechanism the definition of a circular execution must be modified. Removing messages from non active links to construct an infinite execution from $\overline{E}$ as in the proof of Theorem 3.5 may yield an infinite execution in which some processor is enabled only finitely many times. In order to allow repetitions of finite executions to form an infinite fair execution, in which every processor is active infinitely often, we require that each such finite execution contains an atomic step of each processor in the system. For

this we need the concept of a *round* of an execution: Let E' be a minimal prefix of an execution E in which every processor receives a message; E' is the first *round* of E. Let E'' be the suffix of E which satisfies $E = E' \circ E''$. The second round of E is the first round of E'', and so on. Let E_i be the prefix that contains the first i atomic steps of E. Let $t_i = R(E_i)$ be the number of rounds in E_i. The next theorem presents a lower bound for systems equipped with a local time-out mechanism. The proof is similar to the proof of Theorem 3.5.

THEOREM 3.6. *Let PR be a self-stabilizing, message-driven protocol for an arbitrary weak-exclusion task, in a system with a local time-out mechanism. For every execution E of PR, E does not contain a circular sub-execution which contains a complete round. From this we conclude that in each execution of PR, E, the first t rounds contain at least t distinct configurations. Hence, for every $t > 0$, the size of at least one configuration in E_i, is at least $\lceil \log_2(t_i) \rceil$. In particular, in any fair execution, the configuration size is unbounded.*

4. Upper Bound. The *token-passing* task is defined informally as a set of executions in which a single token is present in the entire system and is passed fairly among the system's processors. Token-passing is a special case of mutual-exclusion since possession of the single token can be interpreted as a permission to enter the critical section. For this reason token-passing also satisfies the weak-exclusion property, and hence the lower bound of section 3 holds for it. In particular, it means that any self-stabilizing, message-driven protocol PR for token-passing must use some unbounded resource, since in any infinite execution the system size grows beyond any bound. In this section we present three self-stabilizing, token-passing protocols for systems of two processors. In each protocol the configuration size grows during every execution at a rate that matches the lower bound. Each of these protocols can be easily adapted to work on rings of arbitrary size *without increasing its asymptotic complexity*, by considering the ring as a single virtual link. Similar ideas can be used for adapting the protocols to arbitrary rooted tree systems.

By a standard symmetry argument there exists no self-stabilizing, deterministic, token-passing protocol if the processors are identical. Hence, in this section we assume that the system consists of two distinct processors, called *sender* and *receiver*, connected by two links: The first link carries messages from the sender to the receiver while the second link carries messages from the receiver back to the sender. The receiver processor is identical in all three protocols and it is probably the simplest possible finite-state machine. Its program is to copy each message it receives from its incoming link to its outgoing link without any alteration. To the outside world, the combined behavior of the receiver and the two links looks like the behavior of a single queue whose head and tail are used by the receiver. In our analysis we ignore the receiver and consider systems with a single processor, the sender, communicating with itself using a single link on which messages are kept in FIFO order. In each step the sender consumes a message from the head of the link and puts one (or more) messages back at the tail of the link. Tokens are represented by a special symbol, T, which is appended to some of the messages. Our protocols specify the messages that carry a token, but they do not use explicitly the token symbol T, The protocol should guarantee that eventually there is a unique message in the system to which T is appended. All our protocols assume that initially there is at least one message on the link (this assumption is weaker than both the global and the local versions of the time-out mechanism). With this last assumption, the requirement that the link never becomes empty is equivalent to the requirement that whenever a message is received,

at least one message is sent. Hence in every step of the protocol the sender receives the message on the head of the (single) link and then puts one or more messages at the link's end. The three protocols we present are:

Protocol 1: In this protocol the sender is an infinite state machine, and in every execution the link capacity is unbounded.

Protocol 2: In this protocol the sender is an infinite state machine, but in each infinite execution the link capacity is bounded (the bound for each specific execution depends on its initial configuration).

Protocol 3: In this protocol both processors are finite state machines.

```
1 do forever
2     receive(msg_counter)
3     if msg_counter ≥ counter then (* token arrives *)
4     begin (* send new token *)
5        counter:= msg_counter +1
6        send(counter, T)
7     end
8     else send(counter)
9 end
```

FIG. 1. *protocol 1*

protocol 1 (of the sender) appears in Figure 1. The sender uses a variable called *counter*. Each message consists of the present value of *counter*, possibly with the token symbol T. Whenever the sender receives a message whose counter value, $msg\_counter$, is not smaller than *counter*, it sets $counter := msg\_counter + 1$ and sends this new value of *counter* together with the token T; otherwise the sender just sends the current value of *counter* (without the token T). The token letter T is not used by the protocol itself. The correctness of the protocol is based on the fact that eventually the value of *counter* will be larger than all the values that appear in the messages present on the link in the initial configuration. The asymptotic size of *counter* in each execution is $\Omega(\log t)$, where t is the number of messages sent. The details of the proof are omitted.

4.1. Aperiodic Sequences. Protocols 2 and 3 use the following method: each message is associated with some ternary number which is called *color*. The protocol considers any message whose color is different from the color of the previous message as carrying a token. The sender has a local variable called *token_color*. At any given configuration the sender is sending a sequence of messages whose color is equal to (the value of) *token_color*; at the same time the sender waits for a message whose color is equal to *token_color*. As long as the sender receives messages of different colors it sends messages whose color is equal to *token_color*. Once the sender receives a message whose color is equal to *token_color*, it chooses a new *token_color*, and initiates a new sequence of messages whose color is the new *token_color* by sending the first message in this new sequence. This first message is carrying a (virtual) token. Then the sender continues sending messages of the new *token_color* (without tokens), until it receives a message of the new *token_color*, and so on. Our goal is to reach a configuration after which the link always holds at most two consecutive sequences of messages where the colors of all messages in each sequence are equal. In every step the sender consumes a single message from the first sequence whose color is the previous *token_color* and produces one or more messages whose color is

equal to the present *token_color*. After the last message whose color is the previous *token_color* is consumed the link contains a single sequence of messages whose color is *token_color*. In the next step the sender receives the (single) token carried by this sequence and sends it once again by initiating a new sequence of messages whose color is the new *token_color*. In each of the described configurations there exists a single token which is carried by the first message of the sequence whose color is *token_color*. The correctness of the protocols follows from the fact that the sequences of token-colors sent by the receiver is *aperiodic*, as defined below.

Definition: A sequence $A = (a_1, a_2, \cdots)$ is *periodic* if for some positive integer k and for all $i \geq 1$, $a_i = a_{i+k}$. The sequence A is *eventually periodic* if it has a suffix which is periodic. A is *aperiodic* if it is not eventually periodic.

Aperiodic sequences over the integers $\{0, 1, 2\}$ were used in [1] in order to obtain self-stabilizing, data link protocols. Such sequences are created there either by a random number generator or by an infinite state machine (in the first case the algorithm is randomized). The elements of this sequence are used by the protocol of [1] whenever it has to decide on the ternary number to be sent with a new message. In this paper aperiodic sequences are generated by using a counter and the sequence *xor* defined below:

Definition: For an integer i, $xor(i)$ is the sum of the bits (mod 2) in the binary representation of i (e.g., $xor(1) = xor(2) = 1, xor(3) = 0$). The sequence $(xor(1), xor(2), \cdots)$ is denoted by *xor*.

As we show later, the sequence *xor* is aperiodic.

```
1 do forever
2     receive(color)
3     if color = token_color then (* token arrives *)
4     begin (* send new token *)
5        token_color:= (color +xor(counter) + 1) (mod 3)
6        counter := counter +1
7     end
8     send(token_color)
9 end
```

FIG. 2. *protocol 2*

Protocol 2 (of the sender) which appears in Figure 2, is an improvement of the protocol that appears in [1] in the sense that it achieves the lower bound of the previous section. the sense that it achieves the lower bound of the previous section. (The amount of memory used for producing the aperiodic sequence is not addressed nor specified in [1].) In *protocol 2* the sender keeps a counter in its local memory; whenever a message with a new color is sent the counter is incremented. The new color $\in \{0, 1, 2\}$ is determined by the previous color and by applying *xor* to the counter. Roughly speaking, the correctness of the protocol is implied by the fact that since *xor* is aperiodic, the sequence of *colors* generated by the sender is aperiodic as well. The nature of the variables and the correctness proof of *protocol 2* are easily derived from the description of *protocol 3* and from its correctness proof, hence, they are omitted.

4.2. Informal Description of Protocol 3. We now present *protocol 3*, in which both processors are finite state machines. It is easily observed that when an aperiodic sequence is supplied by some external device, a finite state machine can use

this sequence to perform the protocol in [1]. Our construction uses the fact that the finite state machine augmented with the previously described FIFO link can generate an aperiodic sequence. The finite state machine uses the link both for message-passing and for generating the aperiodic sequence, while its size is kept within the optimal bound. *Protocol 3* can be easily transformed to a self-stabilizing, data link protocol in which both processors are finite state machines.

Protocol 3 appears in Figure 3. In this protocol each message is a pair $(color, bit)$, where $color \in \{0, 1, 2\}$ and $bit \in \{0, 1\}$. The local variables *color* and *token_color* are ternary variables while the local variables *counter_bit, counter_xor, carry,* and *new_counter_bit* are binary. The binary *xor* operation is denoted by $\oplus$. For a sequence $s = (\ (color_1, bit_1), ..., (color_k, bit_k))$ of such messages, $N(s)$ denotes the integer whose binary representation is bit_k, bit_{k-1}, ..., bit_1 (bit_1 is the least significant bit). A maximal sequence of consecutive messages of the same color sent by the sender is called a *block*. For each block b, $N(b)$ denotes the integer described above and $|b|$ denotes the number of messages in b. The first message in each block is viewed as a token. To show that the protocol is self-stabilizing, we have to prove that eventually the link contains exactly one message which is the first message in a block. This goal is achieved by making the sequence of the colors of the blocks aperiodic.

The sender uses a local variable called *token_color*, which denotes the color of the block it is now sending. It continues to send messages of this color as long as the colors of the messages it receives are different from *token_color*. Once the sender receives a message whose color is equal to *token_color* (which eventually means that all messages on the link belong to the same block), it: (a) possibly sends one last message of the current block, (b) changes the value of *token_color*, and (c) sends the first message of a new block, with this new color.

```
1  do forever
2     receive(color,counter_bit)
3     if color = token_color then (* token arrives *)
4       begin
5           if carry = 1 then send (color, 1)
            (* new token *)
6           token_color := (color + counter_xor + 1) (mod 3)
7           counter_xor := 0
8           carry := 1
9       end
10    counter_xor := counter_xor ⊕ counter_bit
11    new_counter_bit := carry ⊕ counter_bit
12    carry := carry ∧ counter_bit
13    send (token_color,new_counter_bit)
14 end
```

FIG. 3. *protocol 3*

In Lemma 4.1 we show that in every execution the sender initiates infinitely many blocks. Let $b_1, b_2 \ldots$ be the sequence of blocks initiated by the sender, where the color of b_i is $color(b_i)$ and the integer it represents is $N(b_i)$, as defined above. The protocol is designed so that the following properties are kept:

(p1) The sequence $(color(b_1), color(b_2), \cdots)$ is aperiodic.

(p2) For every large enough i, $N(b_{i+1}) = N(b_i) + 1$, and the *bit* field in the last message of b_i is 1 (that is: $N(b_i) = i + const$ for some constant *const*, and the representation of $N(b_i)$ by b_i has no leading zeroes, implying that $|b_i| = \lceil \log_2 N(b_i) \rceil$.)

We will prove that *(p1)* above implies that eventually there is only one token in the system, while *(p2)* guarantees that the size of the system is logarithmic in the number of steps. We now show that the protocol indeed satisfies *(p1)* and *(p2)* above. For this, we describe the two rules by which the sender computes the bits and the colors it sends. We need the following definition:

Definition: Let $k \geq 1$. Denote by s_k the sequence of messages whose colors are different from $color(b_k)$, which are received by the sender while it sends the block b_k, and by $N(s_k)$ the integer represented by s_k. Note that s_k consists of one or more complete blocks.

Rule 1: (rule for computing *counter_bit*s): The *counter_bit* sent with each message is sent so that for each k, $N(b_k) = N(s_k) + 1$, and $|b_k| = \max\{|s_k|, \lceil \log_2(N(b_k)) \rceil\}$. In other words: the *counter_bit*s sent in block b_k are obtained by adding 1 to the binary number represented by the messages received while this block is sent.

Rule 2: (rule for computing *token_color*): When receiving a message whose color is equal to the value of *token_color*, the new value of *token_color*, which is the color of the next block, b_{k+1}, is determined as follows: $color(b_{k+1}) = color(b_k) + xor(N(s_k)) + 1 \pmod 3$.

Note that Rule 1 can be implemented by a binary adder which is set to zero at the initiation of each new block, and Rule 2 can be implemented by a counter (mod 2). Thus, both rules are easily implemented by a finite state machine.

4.3. Correctness and Complexity Proofs of Protocol 3. LEMMA 4.1. *In every fair execution, E, the sender initiates an infinite number of blocks.*

Proof. The sender initiates a new block whenever it receives a message whose *color* is equal to the current value of *token_color*. In every atomic step in which the sender receives a message whose *color* is not equal to *token_color*, it sends a message, say M', whose color is *token_color*. Since the link carries messages in FIFO order, the message M is eventually received by the sender and it initiates a new block not later than upon receipt of M. The lemma follows. □

A configuration in an execution is called a *limit configuration* if in the next step of the sender a new *token_color* is computed; that is, the color of the next arriving message is equal to the present value of *token_color*. Observe that at a limit configuration c, the link contains a finite (possibly zero) number of complete blocks, and one possibly incomplete block at the tail of the link (this block may be incomplete since upon receipt of the next message the sender may send one more message in this block, by executing line 5 of the code). The first block has the same color as the last (possibly incomplete) block. For an execution E, we denote by i_k the index of the k-th limit configuration in E. In other words, c_{i_k} is the limit configuration just before b_k is initiated.

Next we prove that the number of blocks in consecutive limit configurations does not increase.

LEMMA 4.2. *Let ℓ_k be the number of blocks in the limit configuration c_{i_k} (including the possibly incomplete block). Then $\ell_k \geq \ell_{k+1}$, with equality only if s_k is a single block.*

Proof. Let $m_k \geq 1$ be the number of blocks in s_k. In the sub-execution starting

with c_{i_k} and ending with $c_{i_{k+1}}$ one block is added to the link (namely, b_k), and m_k blocks of s_k are removed from it. Therefore $\ell_{k+1} = \ell_k + 1 - m_k \leq \ell_k$. □

Next we show that the number of blocks in the limit configurations must eventually get down to one. First we need a technical Lemma:

LEMMA 4.3.

(a) *The sequence xor is aperiodic.*

(b) *Let $(a_1, a_2, \cdots)$ be an eventually periodic sequence, and let $b_i = a_{i+1} - a_i$. Then the sequence $B = (b_1, b_2, \cdots)$ is also eventually periodic.*

(c) *Let $(a_1, a_2, \cdots)$ be an eventually periodic sequence. Then for each $i, p > 0$, the sequence $A(i,p)$=$(a_i, a_{i+p}, a_{i+2p}, \cdots)$ is also eventually periodic.*

Proof.

(a) Assume in contradiction that the sequence $xor = (xor(1), xor(2), \ldots)$ is eventually periodic. Then there exist i and ℓ, s.t. $xor(j) = xor(j + \ell)$ for every $j \geq i$. Let q be a non-negative integer such that $2^q \leq \ell < 2^{q+1}$ and let d be an integer satisfying $d \geq q + 2$ and $2^d \geq i$. Consider the following cases:

- $xor(\ell) = 1$: By the definition of d it holds that $xor(2^d + \ell) = 0$. Thus, $1 = xor(2^d) \neq xor(2^d + \ell) = 0$.
- $xor(\ell) = 0$: Then $xor(\ell) = xor(2^q + \ell) = 0$, and $2^q + \ell < 2^d$. Hence, $xor(2^d + 2^q + \ell) = 1$. Thus, $0 = xor(2^d + 2^q) \neq xor(2^d + 2^q + \ell) = 1$.

Thus, there exist a and b such that: (1) $a > i$ and $b > i$, (2) $a - b = \ell$ and (3) $xor(a) \neq xor(b)$, a contradiction.

(b) This claim is trivial.

(c) Let j and ℓ be such that $xor(k) = xor(k + \ell)$ for every $k \geq j$. Then for every $p > 1$ and $k \geq j$ it holds that $a_k = a_{k+\ell p}$. Thus, the sequence $A(i,p)$ is eventually periodic with period length $\leq \ell$.

□

LEMMA 4.4. *In every fair execution E there exists a suffix in which the number of blocks in the limit configurations is always one.*

Proof. By Lemma 4.2 this number never increases, and hence it eventually remains L for some constant $L > 0$ forever. We shall assume that $L > 1$ and derive a contradiction.

Call a limit configuration c_{i_k} *ultimate* if ℓ_k, the number of blocks in c_{i_k}, is L. If c_{i_k} is ultimate then $\ell_{k+1} = \ell_k$ and hence, by Lemma 4.2, s_k is a single block, which must be b_{k-L}. Thus, the first block that follows s_k is b_{k-L+1}. By the protocol, b_k is terminated when the sender receives a message whose color is equal to the color of b_k. Therefore, we have that the color of (the messages in) the block b_{k-L+1} is equal to the color of the messages in b_k, i.e.: $color(b_{k-L+1}) = color(b_k)$. Hence the sequence $COLORS = (color(b_1), color(b_2), \cdots)$ is eventually periodic with period length $L-1 > 0$. Let $BXOR = (xor(N(b_1)), xor(N(b_2)), \cdots)$. By the way $color(b_{k+1})$ is computed, we have that for an ultimate configuration c_{i_k}, $xor(N(b_k - L)) = [color(b_{k+1}) - color(b_k)]$ (mod 3) -1. Hence, by Lemma 4.3 (b), if $COLORS$ is eventually periodic so is $BXOR$. We shall derive a contradiction by showing that the sequence $BXOR$ is aperiodic.

Lemma 4.3 (c) implies that in order to show that $BXOR$ is aperiodic, it is sufficient to show that for some positive i and p, the sequence $BXOR(i,p) = (xor(N(b_i)), xor(N(b_{i+p})), xor((N(b_{i+2p})), \cdots)$ is aperiodic. For this, observe that for an ultimate configuration c_{i_k}, it must hold that $N(b_k) = N(s_k)+1 = N(b_{k-L})+1$. Hence, for any integer i we have that $BXOR(i,L) = (xor(N(b_i)), xor(N(b_{i+L})), xor((N(b_{i+2L})), \cdots)$ $= (xor(N), xor(N+1), xor(N+2), \cdots)$, where $N = N(b_i)$. Thus, $BXOR(i,L)$ is a

suffix of the sequence *xor*, which is aperiodic by Lemma 4.3 (a). Hence, $BXOR(i, L)$ is also aperiodic. This yields the desired contradiction. □

Lemma 4.4 and its proof imply that properties *(p1)* and *(p2)* hold: Property *(p1)* holds since the proof of Lemma 4.4 shows that the sequence $COLORS$ is aperiodic. Property *(p2)* is proved as follows: Let E' be a suffix of E satisfying Lemma 4.4, and let c_{i_k} be any limit configuration in E'. Then, by Rule 1, $N(b_{k+1}) = N(s_{k+1}) + 1 = N(b_k) + 1$, which easily implies *(p2)*.

We now show that the space complexity of *protocol 3* indeed matches the lower bound of the previous section. Since both the number of states of a processor and the number of distinct messages in our protocol are constants, the size of a configuration is proportional to the number of messages in it. Therefore to bound the size of a configuration from above it is enough to bound the number of messages in it. In the next lemma we show that for each execution $E = (c_0, a_1, c_1, \cdots)$ of the protocol, the size of the i-th configuration of E, c_i, is $O(\log_2(i))$. Let c_{i_k} denotes the k-th limit configuration of E, and let b_k be the corresponding block. We shall prove that $|b_k| = O(\log k)$.

LEMMA 4.5. *For every large enough k, the number of messages in the limit configuration c_{i_k} is $\lceil \log_2 N(b_{k-1}) \rceil$.*

Proof. By Lemma 4.4 there exists a suffix E' of E such that every limit configuration in E' contains one block. Clearly, it is suffices to prove the Lemma for E'. As observed above, property *(p2)* eventually holds for every limit configuration in E'. The lemma follows. □

COROLLARY 4.6. *The number of messages in c_ℓ, the $\ell - th$ configuration of E, is $O(\log_2(\ell))$.*

Proof. Let E' be a suffix of E as in Lemma 4.5, and assume that ℓ is large enough so that c_ℓ belongs to E'. Then the number of messages in c_ℓ is equal to the number of messages in the next limit configuration, c_{i_k}, which is $O(\log_2 k)$ (for some k). The proof is completed by the observation that, since $i_j \geq j$ for all j, and since configuration $c_{i_{k-1}}$ precedes c_ℓ in E, we have that $\ell \geq i_{k-1} + 1 \geq (k-1) + 1 = k$. □

4.4. Larger Systems. Now, we describe how to use our protocols in directed rings with more than two processors. The processors of the ring are denoted by $P_1, \cdots, P_n$ where P_1 is a sender while $P_2, \cdots, P_n$ are receivers. Whenever a processor P_i, $1 < i < n$, receives a message M from P_{i-1}, P_i sends M to P_{i+1}. Similarly, whenever P_n receives a message M from P_{n-1}, it sends M to P_1. Thus, the ring behaves like a virtual link from the sender, P_1, to itself. It is not hard to see that the existence of a single message on the entire ring prevents communication deadlocks, thus, we assume that there is a time-out mechanism that guarantees this condition (this time-out mechanism is invoked only once to recover from initial deadlock configuration). It can be proved, in a way similar to previous proofs, that our protocols guarantee that eventually there is exactly one token that encircles the ring from the sender to itself. Actually, our protocols can be used in any connected system by hardwiring a directed ring that spans the entire system.

4.5. Construction of a Token Controller. In this subsection we define queue machines and token controllers and interpret our results in these terms.

A *queue machine* Q is a finite state machine which is equipped with a queue, which initially contains a non-empty word from Σ^+ for some (finite) alphabet Σ. In each step of its computation Q performs the following: (a) reads and deletes a letter from the head of the queue, (b) adds zero or more letters from Σ to the tail of the

queue, and (c) moves to a new state. The computation terminates when Q halts or when its queue becomes empty, which prevents Q from performing any further steps.

The main difference between queue machines and various types of Turing Machine is that the input alphabet and the work alphabet of a queue machine are identical. For this reason, a queue machine cannot perform simple tasks like deciding the length of the input word, or even deciding whether the input word contains a specific letter[2].

We now define *token controller*, which is a special type of queue machine. Assume that the alphabet Σ contains a specified subset τ of *token letters*. A queue machine is a *token controller* if, starting with a nonempty queue of arbitrary content, eventually the queue contains exactly one occurrence of a letter from τ forever.

A priori, it is not clear that a token controller exists. Observe that if a token controller exists, then its queue never becomes empty (since once the queue is empty it remains so forever). More importantly, a token controller (if exists) can never halt, since it cannot guarantee that upon halting, the queue contains exactly one occurrence of a token letter. The last two observations imply that a token controller can be viewed as a special case of a token-passing system, in which Σ is the set of messages sent by the protocol, and τ is the set of messages that carry the token. We show below how to transform the sender from *protocol 3* to a token controller.

Define the alphabet Σ to be a set of triplets $(color, bit, t)$, where *color* and *bit* are as in *protocol 3*, and t is either T — in case the message carries a token (i.e., it is the first message of some block), or *nil*, in case it does not. The set τ is defined as the set of all possible triplets whose third component is T. The two anti-parallel FIFO links between the sender and the receiver are considered as a single queue. Receiving a message is regarded as deleting a letter from the head of the queue, while sending a message is regarded as appending a message to the end of the queue.

Since *protocol 3* guarantees that eventually exactly one message in every configuration is carrying a token, the queue machine described above is a token controller. Moreover, our lower bound results imply that this token-controller is optimal with respect to the rate in which the size of the queue grows.

5. Self Stabilizing Simulation of Shared Memory. In this section we present a method for simulating self-stabilizing, shared-memory protocols by self-stabilizing, message-driven protocols. The simulated protocols are assumed to be in the shared-memory model defined in [9]. In this model, communication between neighbors, P_i and P_j, is carried out using a two-way link. The link is implemented by two shared registers which support **read** and **write** atomic operations. Processor P_i reads from one register and writes in the other while these functions are reversed for P_j. In the implementing system, every link is simulated by two directed links: one from P_i to P_j and the other from P_j to P_i. The heart of the simulation is a self-stabilizing implementation of the **read** and **write** operations.

The proposed simulation implements these operations by using a self-stabilizing, token-passing protocol. For any pair of neighbors, we run the protocol on the two links connecting them. In order to implement our self-stabilizing, token-passing protocol we need to define for each link which of the processors acts as the sender and which of the processors acts as the receiver. We assume that the processors have distinct identifiers. Every message sent by each of the processors carries the identifier of that processor. Eventually each processor knows the identifier of all its neighbors. In

[2] A variant of queue machine which can use arbitrary work alphabet is in fact an oblivious Turing machine, which is as powerful as a standard Turing machine

each link, the processor with the larger identifier acts as the sender while the other processor acts as the receiver. Since each pair of neighbors uses a different instance of the protocol, a separate time-out mechanism is needed for every such pair. In other words: A correct operation of the simulation requires that for any pair of neighbors there exists at least a single message on one of the two links connecting the neighbors.

We now describe the simulation of some arbitrary link e, connecting P_i and P_j: In the shared memory model, e is implemented by a register $R_{i,j}$ in which P_i writes and from which P_j reads, and by a register $R_{j,i}$ for which the roles are reversed. In the simulating protocol, processor P_i (P_j) keeps a local variable called $r_{i,j}$ ($r_{j,i}$), which keeps the values of $R_{i,j}$ ($R_{j,i}$ respectively). Every token has an additional field called *VALUE*. Every time P_i receives a token from P_j, P_i writes the current value of $r_{i,j}$ in the *VALUE* field of that token. A **write** operation of P_i into $R_{i,j}$ is simply implemented by locally writing into $r_{i,j}$. A **read** operation of P_i from $R_{j,i}$ is implemented by the following steps:

1. P_i receives a token from P_j and then
2. P_i receives another token from P_j. The value read is the *VALUE* attached to the second token.

The correctness of the simulation is proved by showing that for every execution E whose initial configuration contains at least one message on each link, it is possible to linearize all the simulated **read** and **write** operations executed in E so that eventually every simulated **read** operation from $R_{i,j}$ returns the last value that was written to it. (i.e., that the protocol simulates executions in the shared-memory model in which the registers are eventually atomic, see [20]). Define the time of a simulated **write** operation to $R_{i,j}$ to be the time in which the local write operation to $r_{i,j}$ is executed. Define the time of a simulated **read** operation of P_j from $R_{i,j}$ to be the time in which P_i sends the value of its local variable $r_{i,j}$ attached to the token that later reaches P_j in step (2) of the simulated **read**. Once each link holds a single token, all the operations to register $r_{i,j}$ are linearized, and every read operation from $r_{i,j}$ returns the last value written to $r_{i,j}$.

Acknowledgments. We thank Alan Fekete for helpful remarks.

REFERENCES

[1] Y. Afek and G.M. Brown, "Self-Stabilization of the Alternating-Bit Protocol", *Distributed Computing*, 7 (1993), pp. 27-34.

[2] G.M. Brown, M.G. Gouda, and C.L. Wu, "A Self-Stabilizing Token system", *IEEE Transactions on Computers*, 38 (1989), pp. 845-852.

[3] J. Burns, M.G. Gouda and R. E. Miller, "Stabilization and Pseudo stabilization", *Distributed Computing*, 7 (1993), pp. 35-42.

[4] J.E. Burns and J. Pachl, "Uniform Self-Stabilizing Rings", *ACM Transactions on Programing Languages and Systems*, 11 (1989), pp. 330-344.

[5] K.A. Bartlet, R.A. Scantlebury and P.T. Wilkinson, "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links", *Communication of the ACM*, 12 (1969), pp. 260-261.

[6] J.E. Burns, "Self-Stabilizing Rings without Demons", Technical Report GIT-ICS-87/36, Georgia Institute Of Technology.

[7] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control", Communications of the ACM 17,11 (1974), pp. 643-644.

[8] E. W. Dijkstra, "self-stabilizing systems in spite of distributed control (EWD391)", Reprinted in Selected Writing on Computing: A Personal Perspective, Springer-Verlag, Berlin, 1982, pp. 41-46.

[9] S. Dolev, A. Israeli and S. Moran, "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity", *Distributed Computing*, 7 (1993), pp. 3-16.

[10] S. Dolev, A. Israeli and S. Moran, "Resource Bounds for Self Stabilization Message Driven Protocols", *Proc. of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, Montreal, August 1991, pp. 281-294.

[11] S. Dolev, A. Israeli and S. Moran, "Uniform Dynamic Self-Stabilizing Leader Election", Proceedings of the Fifth International Workshop on Distributed Algorithms, Delphi Greece, October 1991, pp. 167-179.

[12] D. Dolev and D. Koller, "Token Survival", preprint.

[13] M.G. Gouda and N.J. Multari, "Stabilizing Communication Protocols", *IEEE Transactions on Computers* , Vol. 40 No. 4 (1991), pp. 448-458.

[14] A. Israeli and M. Jalfon, "Token Management Schemes and Random Walks Yield Self Stabilizing Mutual Exclusion", *Proc. of the Ninth ACM symp. on Principles of Distributed Computing* (1990), pp. 119-131.

[15] A. Israeli and M. Jalfon, "Self-stabilizing Ring Orientation", Proceedings of the Fourth International Workshop on Distributed Algorithms, September 1990, pp. 1-14. Also to appear in Information and Computation.

[16] S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems", *Distributed Computing*, 7 (1993), pp. 17-26.

[17] H.S.M. Kruijer, "Self-stabilization (in spite of distributed control) in tree-structured systems", Information Processing Letters 8,2 (1979), pp. 91-95.

[18] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. of the ACM 21,7*, (1978), pp. 558-565.

[19] L. Lamport, "Solved problems, unsolved problems, and non-problems in concurrency", *Proc. of the Third ACM symp. on Principles of Distributed Computing* (1984), pp. 1-11.

[20] L. Lamport, "On Interprocess Communication. Part I: Basic Formalism", *Distributed Computing 1, 2* 1986, 77-85.

[21] M. Multari, "Toward a Theory for Self-stabilizing Protocols," Ph.D. dissertation, Dep. Comput. Sci. Univ. of Texas at Austin, 1989.

[22] M.Tchuente, "Sur l'auto-stabilisation dans un réseau d'ordinateurs", RAIRO Inf. Theor. 15 (1981), pp. 47-66.