Resource Bounds for Self Stabilizing Message Driven Protocols
(Revised Version of TR 641)

by

S. Dolev, A. Israeli, S. Moran

Technical Report #748
July 1992
Resource Bounds for Self Stabilizing Message Driven Protocols

Shlomi Dolev
Dept. of Computer Science
Technion — Israel

Amos Israeli
Dept. of Electrical Engineering
Technion — Israel

Shlomo Moran
Dept. of Computer Science
Technion — Israel

July 23, 1992

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 results have an interesting interpretation in terms of automata theory.
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, [Dij-74]. The work of [Dij-74] as well as most of the following work on self-stabilizing systems assume the communication model of shared variables. Among these papers are [Kr-79], [Tc-81], [Dij-82], [La-86], [BGW-87], [Bu-87], [BP-88], [IJ-90], [IJ-90a], [DIM-90] and [DIM-91].

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 addressed self-stabilizing, message-passing systems. The work of Katz and Perry, [KP-90], presents a general tool for extending an arbitrary message-passing protocol to a self-stabilizing protocol. The work of Afek and Brown, [AB-89], presents a self-stabilizing version of the well-known alternating-bit protocol, (see e.g. [BSW-69]).

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 Katz and Perry, [KP-90], it is argued 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 be able to 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, and in fact that for every non-trivial, self-stabilizing, message-passing system, in 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. As mentioned before the work of Afek and Brown, [AB-89], presents a self-stabilizing version of a message-driven protocol called the Alternating-bit protocol. The communication deadlock problem is avoided by the use of a time-out mechanism which is an integral part of the original protocol.

In this work we define and study the class of self-stabilizing, message-driven protocols. By the argument of [KP-90], 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 pace we resort to a 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 bound we present takes 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 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 [DK-86] for some fault tolerant message-passing systems and in [IJ-90], 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 $\epsilon$-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, and improves a result of [GM-91], where it is shown that self-stabilizing protocols in such systems must allow infinitely many system configurations (but not that each specific execution must contain infinitely many 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. Similar ideas can be used for adapting the protocol to arbitrary rooted tree systems.

In the first protocol both processors memory and messages size grow unboundedly with time. The second protocol is an improvement on the first protocol in which the size of the memory of the processors grow (in logarithmic rate) while the size of the link content is bounded. The second protocol is an improvement of the deterministic protocol of [AB-89]. 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.
a configuration as above, and let \( a = (i, s_i, (e, msg), (e_1, msg_1), (e_2, msg_2), \ldots, (e_t, msg_t), s_i) \) be an atomic step. \( a \) is applicable to \( (P_i \text{ in} \) \( c \), if \( P_i \) is in state \( s_i \) 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, \( A = (a_1, a_2, \ldots) \), is applicable to a 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, \( E = (c_0, a_1, a_2, \ldots) \), is a (finite or infinite) sequence which starts with a configuration \( c_0 \) and continues with an applicable sequence of atomic steps \( A = (a_1, a_2, \ldots) \). Another representation of an execution is \( E = (c_0, a_1, c_1, a_2, \ldots) \) where for every \( i > 0 \), \( c_{i-1} \xrightarrow{a_i} c_i \). We use each of these two representations wherever it is convenient. A fair execution is an infinite execution in which 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 [La-78]:

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 [La-78]: atomic steps \( a_1, \ldots, a_k \) 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 \( E \). It can be easily shown that for any configuration \( c \), if \( a_1, \ldots, a_k \) are atomic steps which are applicable to \( k \) distinct processors in \( c \), then there exist an execution that starts with \( c \) in which the atomic steps \( a_1, \ldots, a_k \) are concurrent.

An asynchronous protocol, \( PR \), is usually defined by a set of \( n \) processors. An asynchronous protocol defines a set of executions that satisfy the following:

1. If \( E = (c_0, a_1, a_2, \ldots) \) is an arbitrary execution of \( PR \) then every prefix of \( E \), is also an execution of \( PR \).
2. Let \( E = (c_0, a_1, c_1, a_2, \ldots, a_r, c_r) \) be arbitrary finite execution of \( PR \). Then for every atomic step \( a \) such that \( c_r \xrightarrow{a} c \), \( PR \) has an execution \( E \circ (a, c) \).\(^2\)

\(^1\)Unlike previous definitions of message-driven protocols we do not assume that an execution starts with "wakeup" messages, since the system should be able to stabilize starting from any system configuration and in particular a configuration that omit those "wakeup" messages.

\(^2\)For sequences \( S_1 \) and \( S_2 \), \( S_1 \circ S_2 \) denotes the concatenation of \( S_1 \) and \( S_2 \).
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). Put it other way, in proving our lower bounds we assume only that the protocol stabilizes in executions in which no deadlock occurs. For this sake, 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 some *time-out* mechanism. A global time-out mechanism initiates a system in a global deadlock configuration to a default initial configuration. 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*

for every $c$, there is an execution of $PR$ that starts with $c$. Moreover, every such infinite fair execution reaches a safe configuration with respect to $LE$ and $PR$.

Later, we prove a lower bound that holds for systems equipped with a stronger type of time-out mechanism — a local time-out mechanism. An infinite execution $E$ is in a *local* communication deadlock if for some processor $P$, $P$ is activated (i.e. executes an atomic step) finitely many times. 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). Note that a local time-out mechanism is strictly stronger than a global time-out mechanism.

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

for every $c$, there is an execution of $PR$ that starts with $c$. Moreover, 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 \textit{weak-exclusion}. This class contains all non-trivial tasks which require continuous changes in the system's configuration; in particular this class includes both \textit{\ell-exclusion} and \textit{token-passing}. For an execution $E$, denote by $A_{i}(E)$ the set of distinct atomic steps executed by $P_{i}$ during $E$. A task belongs to the \textit{weak-exclusion} class if its set of legal execution, $LE$, satisfies:

\[ [WE] \text{ For any } E \in LE \text{ there exists a set of two or more atomic steps } B = \{a_{i_{1}}, \cdots , a_{i_{k}}\}, \]
\[ 1 \leq i_{1} < \cdots < i_{k} \leq n, \text{ such that the atomic steps in } B \text{ are never simultaneously concurrent during } E. \]

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. Then, we present a slightly weaker lower bound for systems with a local time-out mechanism. 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.

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

**Proposition 1:** For every execution $E = (c_{0}, a_{1}, \cdots , a_{r}, c_{r})$ and for every link $e$, $M_{c}^{e} \circ M_{c,e}^{E} = M_{c}^{e} \circ M_{c,e}^{E}$.

**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 $E = (c_{0}, a_{1}, \cdots , c_{t}, a_{t}, c_{t})$ whose result configuration $c_{t}$ is equal to its initial configuration $c_{0}$ is called a \textit{circular} execution. A link $e$ is \textit{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 $E$ any number of times yields another execution; repeating $E$ forever and removing all occurrences of messages in non active links yields a fair execution. Observe that an execution in which a certain configuration appears more than once has a circular sub-execution, $\overline{E} = (c_{i}, a_{i+1}, \cdots , a_{i+t}, c_{i+t}) \equiv (\overline{c_{0}}, \overline{a_{1}}, \cdots , \overline{a_{t}}, \overline{c_{t}})$. Thus, to show that in every execution of $PR$ all the configurations are distinct, we assume that $PR$ has a circular sub-execution $\overline{E}$ as above, and reach a contradiction by showing that $PR$ is not self stabilizing.

Using $\overline{E}$ we now construct an initial configuration $c_{\text{init}}$ as follows:

- The state of each processor in $c_{\text{init}}$ is equal to its state in $\overline{c_{0}}$. 

6
• For any active link in $E$, $M_{e^{\text{init}}}^c = M_{e^{\text{init}}} \circ M_{e^{\text{res}}}^c$ and for any non active link in $\overline{E}$, $M_{e^{\text{init}}}^c$ is empty.

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

**Lemma 2:** Every $A \in \text{merge}(\bar{A})$ is applicable to $c_{\text{init}}$, and the resulting execution, $E_A = (c_{\text{init}}) \circ A$, is a circular execution of $PR$.

**Proof:** Let $A$ be an arbitrary sequence in $\text{merge}(\bar{A})$ and let $P_i$ be an arbitrary processor of the system. Then we have: (i) The initial state of $P_i$ in $c_{\text{init}}$ is equal to its initial state in $\overline{c_{\text{init}}}$. (ii) In $c_{\text{init}}$ all messages which $P_i$ receives during $E$ 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 $\bar{A}(i)$. (i) - (iii) above imply that the sequence $A$ is applicable to $c_{\text{init}}$, and the application of $A$ to $c_{\text{init}}$ yields an execution, $E_A$, with result configuration, $c_{\text{res}}$, whose state vector is equal to the state vector of $c_{\text{init}}$ and in which for every active link $M_{e^{\text{A}}}^E = M_{e^{\text{E}}}^c$ and $M_{e^{\text{A}}}^{E_A} = M_{e^{\text{E}}}^c$.

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

1. $M_{e^{\text{init}}}^c \circ M_{e^{\text{E}}}^c = M_{e^{\text{E}}}^c \circ M_{e^{\text{res}}}^c$ (by Proposition 1 and by the fact that $M_{e^{\text{A}}}^E = M_{e^{\text{E}}}^c$ and $M_{e^{\text{A}}}^{E_A} = M_{e^{\text{E}}}^c$).

2. $M_{e^{\text{in}}}^c \circ M_{e^{\text{E}}}^c = M_{e^{\text{E}}}^c \circ M_{e^{\text{res}}}^c$ (by Proposition 1 and the circularity of $E$).

Replacing $M_{e^{\text{init}}}^c$ in equation 1 with its explicit contents yields:

3. $M_{e^{\text{in}}}^c \circ M_{e^{\text{E}}}^c \circ M_{e^{\text{res}}}^c$.

Using equation 2 to replace $M_{e^{\text{in}}}^c \circ M_{e^{\text{E}}}^c$ by $M_{e^{\text{E}}}^c \circ M_{e^{\text{res}}}^c$ in equation 3 gives:

4. $M_{e^{\text{E}}}^c \circ M_{e^{\text{res}}}^c$.

Dropping $M_{e^{\text{E}}}^c$ from the two sides of equation 4 yields the desired result: $M_{e^{\text{init}}}^c = M_{e^{\text{res}}}^c$, which proves the lemma.

Define $\text{blowup}(E)$ to be the set of executions consisting of $c_{\text{init}}$ as initial configuration and any $A \in \text{merge}(\bar{A})$ as the sequence of atomic steps. Notice that, for every circular execution $E$ and for every execution $E \in \text{blowup}(E)$ it holds that $A_i(E) = A_i(E)$. 

7
Lemma 3: For any set of atomic steps $B = \{a_{i_1}, \cdots, a_{i_k}\}$, $1 \leq i_1 < \cdots < i_k \leq n$, where $a_{i_j} \in A_{i_j}(E)$, there is an execution $E \in blowup(E)$ that contains a configuration for which all the atomic steps in $B$ are concurrent simultaneously.

Proof: For notational simplicity, assume that $k = n$ and that $B = \{a_1, a_2, \cdots, a_n\}$. Let $A \in merge(A_i)$ be the sequence constructed as follows: first take all the steps in $A_i(1)$ that precede $a_1$, then take all the steps in $A_i(2)$ that precede $a_2$, ..., then take all the steps in $A_i(n)$ 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 $merge(A)$ by taking the remaining atomic steps in an arbitrary order, which keeps the internal order of each $A_i$.

Corollary 4: Let $E \in blowup(E)$ and let $E' = E^\infty$ (i.e., $E'$ is obtained by repeating $E$ forever). Then $E'$ is fair, and no configuration in $E'$ is safe for the protocol $PR$.

Proof: By the definition of a circular execution and by the definition of $blowup(E)$, $E'$ is fair. So it remains to show that no configuration in $E'$ is safe.

Assume by way of contradiction that some configuration $c$ in $E'$ is safe. Since $PR$ is a protocol for some weak-exclusion task, there exists some set of atomic steps $B = \{a_{i_1}, \cdots, a_{i_n}\}$ for which there is no computation $E''$ that starts from $c$ in which all the $a_{i_j}$'s are applicable simultaneously. We reach a contradiction by showing that the last statement is incorrect. An execution $E''$ which contradicts that statement is constructed by first continuing the computation from $c$ as in $E$ until the configuration $c_{init}$ is reached. Then apply Lemma 3 to extend $E''$ by a computation $E_A$ which contains a configuration in which all the $a_{i_j}$'s are applicable.

Note: The construction in Corollary 4 can be easily modified to yield an execution $E$ which does not have any suffix in $LE$, and hence the protocol $PR$ is not even pseudo self stabilizing (see [BGM-90]).

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

Theorem 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 Corollary 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 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 $E' \circ E'' = E$. The second round of $E$ is the first round of $E''$, and so on.

Let $PR$ be a message-driven, self-stabilizing protocol for a weak-exclusion task, in a system with a local time-out mechanism. It is not hard to show, by arguments similar to the ones used in the proof of Theorem 5, that no execution of $E$ contains a circular sub-execution which contains at least one round. From this we conclude that in each execution of $PR$, the first $t$ rounds contain at least $t$ distinct configurations. The proof is concluded with the following corollary:

**Corollary 6:** Let $E$ be an execution of a self-stabilizing weak-exclusion protocol, in a system with a local time-out mechanism. 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 $E_i$. Then there exists a configuration in $E_i$ whose size is at least $\lceil \log_2(t_i) \rceil$. In particular, in any fair execution of $E$ 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 anti-parallel links. The receiver simply returns any message it receives back to the sender. Since the protocol is entirely determined by the algorithm of the sender we ignore the receiver and consider the two anti-parallel links as a single link on which messages are kept in FIFO order. 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 $msg\_counter \geq counter$ then (* token arrives *)
4 begin (* send new token *)
5 \hspace{1cm} counter := $msg\_counter +1$
6 \hspace{1cm} send($counter, T$)
7 end
8 else send($counter$)
9 end

![Figure 1: protocol 1](image)

**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 \textit{token\_color}. At any given configuration the sender is sending a sequence of messages
whose color is equal to (the value of) \textit{token\_color}; at the same time the sender waits for
a message whose color is equal to \textit{token\_color}. As long as the sender receives messages of
different colors it sends messages whose color is equal to \textit{token\_color}. Once the sender
receives a message whose color is equal to \textit{token\_color}, it chooses a new \textit{token\_color}, and
initiates a new sequence of messages whose color is the new \textit{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 \textit{token\_color} (without tokens), until it receives
a message of the new \textit{token\_color}, and so on and so forth. 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
messages from the first sequence whose color is the previous \textit{token\_color} and produces one
or more messages whose color is equal to the present \textit{token\_color}. After the last message
whose color is the previous \textit{token\_color} is consumed the link contains a single sequence of
messages whose color is \textit{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 \textit{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 \textit{token\_color}.
The correctness of the protocols follows from the fact that the sequences of token-colors sent
by the receiver is \textit{aperiodic}, as defined below.

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

Aperiodic sequences over the integers \( \{0, 1, 2\} \) were used in [AB-89] 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 [AB-89] 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 \textit{xor} defined below:

\textbf{Definition:} For an integer \( i \), \textit{xor}(i) is the sum of the bits (mod 2) in the binary repre-
sentation of \( i \) (e.g., \( \text{xor}(1) = \text{xor}(2) = 1, \text{xor}(3) = 0 \)). The sequence \( (\text{xor}(1), \text{xor}(2), \cdots) \) is
denoted by \textit{xor}.

As we show later, the sequence \textit{xor} is aperiodic.

\textit{Protocol 2} (of the sender) which appears in Figure 2, is an improvement of the protocol
that appears in [AB-89] in the sense that it achieves the lower bound of the previous section.
In \textit{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 \textit{xor} to the counter. The nature of the variables and the
correctness proof of \textit{protocol 2} are easily derived from the description of \textit{protocol 3} and from
its correctness proof, and hence are omitted.
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

Figure 2: protocol 2

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 [AB-89]. Our construction uses the fact that the finite state machine augmented with the single FIFO link which we described before can generate an aperiodic sequence. The subtle part in the protocol is in combining the construction of the aperiodic sequence and its use to be done simultaneously by a finite state machine. The finite state machine uses the link both for passing the messages and for generating the sequence, while keeping its size 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 variables counter_bit, counter_xor, carry, and new_counter_bit are binary. The binary xor operation is denoted by \( \oplus \). For a sequence \( s = ((color_1, bit_1), \ldots, (color_k, bit_k)) \) of such messages, \( N(s) \) denotes the integer whose binary representation is \( bit_k, bit_{k-1}, \ldots, bit_1 \) (\( bit_1 \) 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.

In Lemma 7 we show that in every execution the sender initiates infinitely many blocks.
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)
6             (* new token *)
7          token_color := (color + counter_xor + 1) (mod 3)
8          counter_xor := 0
9          carry := 1
10         end
11 new_counter_bit := carry * counter_bit
12 carry := carry \land counter_bit
13 send (token_color, new_counter_bit)
14 end

Figure 3: protocol 3

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 $|b_i| = \lfloor \log_2 N(b_i) \rfloor$.)

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_bits): The counter_bit sent with each message is sent so that for each $k$, $N(b_k) = N(s_k) + 1$, and $|b_k| = \max\{|s_k|, \lfloor \log_2 (N(b_k)) \rfloor \}$. In other words: the counter_bits 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
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 \( \text{xor} = (\text{xor}(1), \text{xor}(2), \ldots) \) is eventually periodic. Then there exist \( i \) and \( \ell \), s.t. \( \text{xor}(j) = \text{xor}(j + \ell) \) 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:

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

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

(b) This claim is trivial.

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

**Lemma 10:** 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 8 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 \( \ell_k \), 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 8, \( s_k \) 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.: \( \text{color}(b_{k-L+1}) = \text{color}(b_k) \). Hence the sequence \( \text{COLORS} = (\text{color}(b_1), \text{color}(b_2), \cdots) \) is eventually periodic with period length \( L-1 > 0 \). Let \( BXOR = (\text{xor}(N(b_1)), \text{xor}(N(b_2)), \cdots) \). By the way \( \text{color}(b_{k+1}) \) is computed, we have that for an ultimate configuration \( c_{i_k} \), \( \text{xor}(N(b_{k-L})) = [\text{color}(b_{k+1}) - \text{color}(b_k)] (\text{mod} 3) - 1 \). Hence, by Lemma 9 (b), if \( \text{COLORS} \) is eventually periodic so is \( BXOR \). We shall derive a contradiction by showing that the sequence \( BXOR \) is aperiodic.

 Lemma 9 (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 \( BXOR(i, p) = (\text{xor}(N(b_i)), \text{xor}(N(b_{i+p})), \text{xor}(N(b_{i+2p})), \cdots) \) 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 \( BXOR(i, L) = (\text{xor}(N(b_i)), \text{xor}(N(b_{i+L})), \text{xor}(N(b_{i+2L})), \cdots) = \)
(xor(N), xor(N + 1), xor(N + 2), ...), where \( N = N(b_i) \). Thus, \( BXOR(i, L) \) is a suffix of the sequence \( xor \), which is aperiodic by Lemma 9 (a). Hence, \( BXOR(i, L) \) is also aperiodic. This yields the desired contradiction. \( \square \)

Lemma 10 and its proof imply that properties \((p1)\) and \((p2)\) hold: Property \((p1)\) holds since the proof of Lemma 10 shows that the sequence \( COLORS \) is aperiodic. Property \((p2)\) is proved as follows: Let \( E' \) be a suffix of \( E \) satisfying Lemma 10, and let \( c_{ik} \) 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 \( E = (c_0, a_1, c_1, \ldots) \) of the protocol, the size of the \( i \)-th configuration of \( E \), \( c_i \), is \( O(\log_2(i)) \). Let \( c_{ik} \) 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 11:** For every large enough \( k \), the number of messages in the limit configuration \( c_{ik} \) is \( \lfloor \log_2 N(b_{k-1}) \rfloor \).

**Proof:** By Lemma 10 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. \( \square \)

**Corollary 12:** The number of messages in \( c_{i_\ell} \), the \( \ell \)-th configuration of \( E \), is \( O(\log_2(\ell)) \).

**Proof:** Let \( E' \) be a suffix of \( E \) as in Lemma 11, and assume that \( \ell \) is large enough so that \( c_{i_\ell} \) belongs to \( E' \). Then the number of messages in \( c_{i_\ell} \) 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 \( c_{i_\ell} \) in \( E \), we have that \( \ell \geq i_{k-1} + 1 \geq (k - 1) + 1 = k \). \( \square \)

### 4.4 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 \( \Sigma^+ \) for some (finite) alphabet \( \Sigma \). 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 \( \Sigma \) 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\(^3\).

We now define *token controller*, which is a special type of queue machine. Assume that the alphabet \(\Sigma\) contains a specified subset \(\tau\) 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 \(\tau\) 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 systems, in which \(\Sigma\) is the set of messages sent by the protocol, and \(\tau\) 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 \(\Sigma\) to be a set of triplets \((\text{color}, \text{bit}, t)\), where \text{color} and \text{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 \(\text{nil}\), in case it does not. The set \(\tau\) 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 the 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 [DIM-90]. Similar to the message-passing model, communication between every pair of neighboring processors in the shared-memory model is carried out using a two-way link. The link however is implemented by two shared registers which support \textbf{read} and \textbf{write} atomic operations. One of the neighbors reads from one register and writes in the other while these functions are reversed for the other neighbor. The heart of the simulation is a self-stabilizing implementation of the \textbf{read} and \textbf{write} operations.

The proposed simulation implements these operations by using a self-stabilizing, token-passing protocol. We run the protocol on any link \(e\) (that stands for two directed anti-parallel

\(^3\)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.
In order to implement our self-stabilizing, token-passing protocol we need to define for any 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 each link the processor with the larger identifier acts as the sender while the other processor acts as the receiver.

We now describe the simulation of some arbitrary link $e$, connecting $P_i$ and $P_j$, by the token-passing protocol: In the shared memory model $e$ is implemented by a register $R_{i,j}$ 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 $R_{i,j}$ ($R_{j,i}$ resp.). 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 order in time 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 [La-86]). 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 totally ordered in time, and every read operation from $r_{i,j}$ returns the last value written to it.

Acknowledgments

We thank Allen Fekete for helpful remarks.

References


