RESOURCE BOUNDS FOR SELF STABILIZING MESSAGE DRIVEN PROTOCOLS

by

S. Dolev, A. Israeli, and S. Moran

Technical Report #641

July 1990
Resource Bounds for Self Stabilizing Message Driven Protocols

(Extended Abstract)

Shlomi Dolev
Dept. of Computer Science
Technion—Israel

Amos Israeli
Dept. of Electrical Engineering
Technion — Israel

Shlomo Moran
Dept. of Computer Science
Technion — Israel

24 May 1990

Abstract

A self-stabilizing system is a distributed system which 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.

Most of the work in this field assumes the communication model of shared variables. The paradigm of self-stabilization is a very general one and does not depend on the communication media used by the system’s processors. A very natural subject would be to look at self-stabilizing message passing systems. Surprisingly, there are very few papers which addressed this subject.

In this work the class of self stabilizing message driven protocols is defined and studied. The viability of this new class is demonstrated by proving non-trivial lower and upper bounds on the resources required by self stabilizing message driven protocols for token passing.

The main technical contribution of this paper is a self stabilizing token passing protocol in which processors are finite state machines and (hence) messages are of fixed size. Our results have an interesting interpretation in terms of automata theory.
1 Introduction

A self-stabilizing system is a distributed system which 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 processor 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.

The study of self-stabilizing systems started with the fundamental paper of Dijkstra, [Di-74]. This paper and most of the following work in this field assume the communication model of shared variables, among these papers are [Kr-79], [TcSI], [Di-S2], [Di-S6], [La-S6], [BGW-87], [Bu-87], [BP-88], [DIM-89] and [JU-89].

The paradigm of self-stabilization is a very general one and does not depend on the communication media used by the system's processes. A very natural subject would be to look at self-stabilizing message passing systems. In particular, it is customarily assumed that message might be corrupted over links, hence processors may enter arbitrary states and link contents may be arbitrary. Self stabilizing protocols treat this problem in a natural way. Surprisingly, there are very few papers which addressed this subject.

A message passing system is a set of state machines, called processors, which communicate by message passing along some communication links. A configuration of any message passing system consists of the state of each processor and 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.

The work of Katz and Perry, [KP-89], presents a general tool for extending an arbitrary message passing protocol, to a self stabilizing protocol. In this work 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, therefore a completely asynchronous self stabilizing system cannot be message driven. As a result it is shown in [KP-89] that in any self stabilizing message passing system, the rate of growth of the configuration size cannot be bounded from above.

The work of Afek and Brown, [AB-89], presents a self stabilizing version of the well-known token passing protocol, (see e.g. [BSW-69]). This is a message driven protocol in which the content of the link does not grow. In return the processors used are either infinite state machines or finite state machines augmented with a random number generator. 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 new class of self stabilizing message driven protocols. We view this novel kind of protocols as very useful in the design and analysis of self stabilizing message passing protocols. After presenting the new class we demonstrate its viability by proving non-trivial lower and upper bounds for self stabilizing message driven protocols for token passing. Informally, the token passing task is to reach a system configuration in which exactly one token is present in the entire system and then to pass the single token fairly among the system's
shared memory systems. The token passing task can be looked at as a special case of mutual
processors. This is a very basic task in fault tolerant systems, among other works it was studied
in [DK-86] for some fault tolerant message passing system and in [IJ-89] for self stabilizing
shared memory systems. The token passing task can be looked at as a special case of mutual
exclusion since having the single token can be interpreted as a permission to enter the critical
section.

We proceed by proving a lower bound on the configuration size for protocols for a large
class of tasks called weak exclusion. To the best of our knowledge this is the first time that a
nontrivial lower bound is proved for such systems. The weak exclusion class contains all non-
trivial tasks which require continuous changes in the system's configuration; in particular this
class includes both mutual 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 holds trivially for the model considered by [KP-89]). This lower bound does not specify
which part of the system grows, it is 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 examine upper bounds for self stabilizing message driven protocols for token passing.
It turns out that for a two processor system it is enough to assume that at least a single message
is present on some communication link to achieve message driven self stabilizing protocols for
token passing with no communication deadlock. 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 can easily be adapted to work
on any arbitrary ring without increasing their asymptotic complexity, by considering the ring as
a single virtual link. Similar ideas can be used for adapting the protocol to arbitrary rooted tree
systems.

The first protocol is designed along the lines presented in [KP-89]. In this protocol both
processor configuration and message size grow unboundedly with time. The second protocol is
an implementation of the first protocol in which processors grow (very slowly) while the size of
the link content remains constant. The deterministic protocol of [AB-89] has similar properties.
The first two protocols are brought for completeness of the presentation.

The main technical contribution of this paper is the third protocol. This 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. This protocol subsumes the two
previous protocols.

Our results can be described also in the terms of automata theory, as follows: Let \( \Sigma \) be a
given alphabet. Define a queue machine \( Q \) to be a finite state machine which is equipped with a
queue, which initially contains a non empty word from \( \Sigma^+ \). 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 to the tail of the queue, and (c) moves to a new state. It
is worth while to note that queue machine appears to be similar to one tape oblivious Turing
machine (that is: a Turing machine that repeatedly scans the nonblank portion of the tape from
left to right), with the initial content of the queue being the input word. However, there are two
differences between these models: The first difference lies in the fact that the queue machine


is allowed any initial state while an oblivious Turing machine assumes a single distinguished initial state. The second, and more crucial, difference is due to the fact that the input alphabet and the work alphabet of the queue machine are identical, while an oblivious Turing machine may use an arbitrary work alphabet. This difference is demonstrated by the fact that oblivious Turing machine, even when initiated to arbitrary state, is as powerful as a standard Turing machine. On the other hand, a queue machine, even when initiated to a specific initial state, cannot perform simple tasks like deciding 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 \( \tau \) of token letters. A queue machine is a queue controller if, starting with a nonempty queue of arbitrary content, eventually the queue contains exactly one occurrence of a letter from \( \tau \) forever. Our lower bound result implies that if a queue controller exists, then in every computation the size of the queue must grow at least logarithmic in the number of moves of the machine. The upper bound implies that a queue controller indeed exists. In view of the fact that a queue machine cannot compute any estimation of the number of occurrences of letters from \( \tau \) 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. Each processor is a state machine which resides on a distinct node of the system's communication graph \( G(V,E) \). Processors communicate using message passing along links. An edge \( e = (i,j) \) of \( G \) stands for two directed antiparallel 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 arrive their destination yet, are stored on the link and transferred in FIFO (First In First Out) order. The graph \( G \) is not required to be simple, we allow more than a single link between two processors.

Whenever a processor is active it executes an atomic step. In a message driven protocol an atomic step of any processor \( P \) begins with a receive action in which \( P \) receives a message from one of its incoming links. The atomic step ends with zero or more send actions in which \( P \) sends messages along some of its outgoing links. An atomic step of \( P_i \) is denoted by \( a = (i,(e, \text{msg}), (e_1, \text{msg}_1), (e_2, \text{msg}_2) \cdots (e_i, \text{msg}_i)) \) where \( e \) is the link through which \( P_i \) receives the message \( \text{msg}_i \), and \( e_1, e_2, \cdots, e_i \) are the outgoing links along which \( P_i \) sends \( \text{msg}_1, \text{msg}_2, \cdots, \text{msg}_i \) respectively.

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 \( c = (s_1 \times s_2 \times \cdots s_n \times M_{e_1} \times M_{e_2} \cdots \times M_{e_m}) \) 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 \( 1 \leq j \leq m \).

We assume that at any given time exactly one atomic step is executed in the entire system. Let \( c \) be a configuration as above, and let \( a = (i,(e, \text{msg}), (e_1, \text{msg}_1), (e_2, \text{msg}_2) \cdots (e_i, \text{msg}_i)) \) be
an atomic step. \(a\) is applicable to \(c\), if whenever \(P_i\) is in state \(a_i\) and it receives \(msg\) along link \(e\), \(P_i\) sends messages \(msg_1, msg_2, \cdots, msg_i\) on its outgoing links \(e_1, e_2, \cdots, e_i\) respectively. In addition \(msg\) should be 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 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 and so forth. An execution, \(E = (c_0, a_1, a_2, \cdots)\), is a sequence which consists of a configuration \(c_0\) and an applicable sequence of atomic steps \(a_1, a_2, \cdots\). Another representation of an execution is \(E = (c_0, a_1, a_2, \cdots)\) where for every \(i > 0\), \(c_{i-1} \xrightarrow{a_i} c_i\). We use each of these two representations wherever its convenient.

A fair execution is an execution in which each message which is either present in the initial configuration or sent during the execution is received. This requirement guarantees that each message is delayed on its link for a finite time.

### 2.2 Self Stabilizing Message Driven Protocols

A self stabilizing system should demonstrate legal behavior some time after it is started from any arbitrary configuration. A behavior of a system is specified by a set of executions. Define a task \(LE\) to be a set of executions which are called legal executions. A configuration \(c\) is safe with respect to a task \(LE\) and a protocol \(PR\) if any fair execution of \(PR\) starting from \(c\) belongs to \(LE\). Note that once the system reaches a safe configuration every subsequent configuration is also safe. Define now the self stabilization requirement as:

**[Self Stabilization]** A protocol \(PR\) is self stabilizing with respect to a task \(LE\) if every fair execution of \(PR\), starting from any configuration, contains a safe configuration with respect to \(LE\) and \(PR\).

For purposes of proving lower bound results on self stabilizing message driven protocols we assume the following (very strong) time-out mechanisms:

**global version:** Whenever the system reaches a communication deadlock (that is: no messages are present on any link and hence, since the protocol is message driven, no processor is about to send a message) the system is initiated to some default initial configuration.

**local version:** Whenever the initial configuration of an execution can be extended to a fair execution in which some processor is never activated, the system is initiated to some default initial configuration.

It is also assumed that the mechanism above is not applicable to any configuration that is reachable from the default initial configuration.

In this extended abstract we consider the global version. Treatment of the local version yields a slightly weaker lower bound and is deferred to the full paper.

### 3 Lower Bound

Denote by \(INF_i(E)\) the set of states of \(P_i\) which occur infinitely often during an infinite execution \(E\). \((INF_i(E) \subseteq S_i)\) The task **Weak Exclusion** is defined to be the set of legal executions, \(LE\),
which satisfies:

[Weak Exclusion] For any $E \in LE$ there exist two states $s_i \in INF_i(E), s_j \in INF_j(E), i \neq j$, such that $s_i$ and $s_j$ do not appear simultaneously in any configuration of $E$.

It is easily observed that every mutual exclusion protocol is also a weak exclusion protocol (the opposite is not true). In this section we establish a lower bound on the size of configurations of any self-stabilizing protocol for the weak exclusion task. In this extended abstract we consider the global version of the aforementioned time-out assumptions. Treatment of the local version yields a slightly weaker lower bound and is deferred to the full paper.

An execution $E = (c_0, a_1, \ldots, c_{l-1}, a_l)$ 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 message are received (and hence by the circularity of $E$ some messages are sent) along $e$ in $E$. Concatenating a circular execution $E$ to itself any number of times yields another execution; the result of concatenating $E$ to itself infinitely often and removing all messages from non active links is a fair execution.

**Theorem 1**: Let $PR$ be a self stabilizing message driven protocol for the weak exclusion task. If $E$ is an arbitrary execution of $PR$ then all configurations of $E$ are distinct and hence for any $t > 0$ the size of at least one of the first $t$ configuration is $\Omega([\log(t)])$.

**Sketch of Proof**: The proof of Theorem 1 uses the fact that any execution which does not satisfy the assumptions of the theorem has a circular subexecution, $E_{circ}$. Using $E_{circ}$ we construct a set of circular executions, $\text{blowup}(E_{circ})$. All circular executions in $\text{blowup}(E_{circ})$ have the same initial configuration $c_{init}$ which is not safe with respect to the weak exclusion task. Configuration $c_{init}$ is obtained from the initial configuration of $E_{circ}$ by blowing up the link contents of all active links in some specific way, and removing all messages from non active links. The proof is completed by the observation that any infinite concatenation of executions from $\text{blowup}(E_{circ})$ is a fair execution which contains $c_{init}$ infinitely often.

### 4 Upper Bound

In this section we define the token passing task and present self stabilizing systems which implement this task and whose configuration size matches the lower bound presented in section 3. Informally, The token passing task is defined 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. The token passing task can be looked as a special case of mutual exclusion since having the single token can be interpreted as a permission to enter the critical section. For this reason the token passing task also satisfies the weak exclusion property, and hence the lower bound of section 3 applies for it. In particular, it means that any self stabilizing message driven protocol $PR$ for this task must use some unbounded resource, since in any infinite execution the system size must grow beyond any bound.

Motivated by the lower bound we present three self stabilizing token passing protocols for two processors whose configuration size matches the lower bound. All protocols can easily
be adapted to work on any arbitrary ring without increasing their asymptotic complexity, by considering the ring as a single virtual link. Similar ideas can be used for adapting the protocol to arbitrary rooted tree systems. The protocols we present are:

- **protocol 1**: This protocol assumes that the processors are infinite state machines, and in each infinite run the link capacity is unbounded.

- **protocol 2**: This protocol assumes that one processor is an infinite state machine, but in each infinite run the link capacity is bounded (the bound for each specific run depends on the initial configuration it starts from).

- **protocol 3**: This protocol assumes that both processors are finite state machines.

While protocols 1 and 2 are relatively straightforward to demonstrate, the existence of protocol 3 is rather surprising, and is implied by a construction of a finite state machine that controls queues of arbitrary size and content - a result which seems to be of independent interest. All three protocols assume that initially there is at least one message in the system (this assumption is weaker than both global and local versions of the time-out mechanism).

By a standard symmetry argument there is 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*. In all three protocols, the receiver just returns any message it receives back to the sender. Thus, the protocol is uniquely determined by the algorithm of the sender. The token is represented by a special symbol, $T$, which is appended to some of the messages. The goal of the protocol is to guarantee that eventually there is a unique message in the system to which $T$ is appended.

**protocol 1** (of the sender) appears in Fig 1. It 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 its local variable *counter*, it sets $counter := msg\_counter + 1$ and sends this new value of *counter* together with the token $T$; otherwise it just sends the current value of *counter* (without the token $T$). 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 in the initial configuration. The asymptotic size of *counter* is $O(\log t)$, where $t$ is the number of messages sent. (Similar ideas are used in [KP-89]). The details of the proof are omitted.

**protocol 2** is similar to **protocol 1**, with the following modification: In order to keep the number of bits on the link bounded, the sender sends the counter bit by bit. In order to distinguish between bits of consecutive values of the counter, each bit is associated with a *color*, which is either $c_0$ or $c_1$. The color of the bit sent is different from the color of the bit received (this idea is similar to that of the alternating bit protocol). In this protocol the sender starts sending the bits of the next number before it complete receiving the previous one. When the sender completes receiving a number, it decides on the exact value of the next number, and whether to send a token with it (the token is sent with the last bit of the number). The details of this protocol are omitted from this version.
from the head of the queue, (b) adds one or more messages to the tail of the queue, and (c) letter taken from some finite alphabet. Thus, the sender can be viewed as a finite state machine.

The subtle part in the protocol is in combining the machines which are augmented by a queue. The self stabilizing token passing protocol uses aperiodic sequences, defined below.

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

Aperiodic sequences were used in [AB-89] in order to achieve 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 resulted algorithm is randomized), and the elements of this sequence are used by the protocol whenever it has to decide on the ternary number to be sent with a new message. 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 observation that aperiodic sequences can be generated by finite state machines which are augmented by a queue. The subtle part in the protocol is in combining the construction and the application of the aperiodic sequence to be done simultaneously by a finite state machine, that uses the link both for passing the messages and for generating the sequence, while keeping its size within the optimal bound. Our protocol can easily be transformed to a self stabilizing data link protocol in which both processors are finite state machines.

As in the previous two protocols, the algorithm of the receiver is just to send the messages it receives back to the sender. Therefore, we can ignore the receiver and view the links between the receiver and the sender as a single queue. In a single step, the sender (a) receives a message from the head of the queue, (b) adds one or more messages to the tail of the queue, and (c) moves to a new state. Since the sender is a finite state machine, there is only a finite number of different messages it can send (and receive), and hence each such message can be considered as a letter taken from some finite alphabet. Thus, the sender can be viewed as a finite state machine $Q$ satisfying (a) - (c) above, where the messages are letters from some alphabet. Initially $Q$ is in an arbitrary state, and the queue contains a nonempty word of arbitrary length and content.

Every move of $Q$ changes the content of the queue. The self stabilizing token passing protocol must satisfy the following properties: starting with arbitrary nonempty queue, the queue never

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

Figure 1: protocol 1
becomes empty (i.e., there is no communication deadlock), and eventually, the queue contains exactly one occurrence of a letter from a well defined set of token letters, \( \tau \), forever (i.e., there is a single token in the system). The requirement that the queue never becomes empty is equivalent to the requirement that whenever a message is received, at least one message is sent. Our protocol implies the existence of queue controllers, as defined in the introduction, for which the rate of growth of the queue is minimized.

Protocol 3 appears in Fig 2. In this protocol each message is a pair \((c, b)\), where \(c\) is a color: \(\{0, 1, 2\}\) and \(b\) is a counter-bit. For a sequence \(S = ((c_1, b_1), \ldots, (c_k, b_k))\) of such messages, \(N(s)\) denotes the integer whose binary representation is \(b_kb_{k-1}\ldots b_1\) (\(b_i\) 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. \(^1\) To show that the protocol works, we have to prove that eventually the queue 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 an aperiodic sequence.

### 4.1.1 Informal description of protocol 3

The sender uses a local variable \textit{expected.color}, which denotes the color of the block it is now sending. It continue to send messages of this color as long as the colors of the messages it receives are different from \textit{expected.color}. Once it receives a message whose color equals \textit{expected.color} (which, hopefully, will eventually mean that all messages on the link belong to the same block), it:

(a) possibly send one last message of the current block, (b) changes the value of \textit{expected.color}, and (c) sends the first message of a new block, with this new color.

In Lemma 2 below 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 \(c(b_i)\) and it represents the integer \(N(b_i)\), as defined above. The protocol is designed so that the following properties are kept:

1. The sequence\((c(b_1), \ldots, c(b_i), \ldots)\) is aperiodic.

2. For all large enough \(i\), \(N(b_{i+1}) = N(b_i) + 1\), and the leading bit in \(b_i\) is 1 (that is: \(N(b_i) = \log_2 N(b_i)\cdot j\) for some constant, and \(|b_i| = \log_2 N(b_i)\)).

We will prove that (1) above implies that eventually there is only one token in the system, while (2) guarantees that the size of the system is logarithmic in the number of steps. We now show that the protocol indeed satisfies (1) and (2) above. For this, we describe the two rules by which the sender decides the bits and the colors it sends. We need two more definitions:

**Definition:** For an integer \(i\), \(\text{xor}_i\) is the parity of the number of ones in the binary representation of \(i\) (e.g., \(\text{xor}_1 = \text{xor}_2 = 1, \text{xor}_3 = 0\)).

\(^1\)To account formally to the requirements of a queue controller we have to define each queue letter (message) as a triplet whose first two components are \((c, b)\) while the third component is either \(T\) (in case the message carries a token or \(\text{nil}\) in case it does not). The set \(c\) is defined as the set of all possible triplets whose third component is \(T\). Since the protocol does not use this third component we omit it from the description.
1 do forever
   2 receive(color, counter_bit)
   3 if color = expected_color (* token returns *) then
      4 begin
         5 if carry = 1 then send (color, 1)
         6 expected_color := (color + counter_xor + 1) (mod 3) (* new token *)
         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 (expected_color, new_counter_bit)
end (expected_color, new_counter_bit)

Figure 2: protocol 3

Definition: $s_k$ is the sequence of messages of color $\neq c(b_k)$, which were received by the sender while it sent the block $b_k$, and $N(s_k)$ is the integer represented by $s_k$. Note that $s_k$ consists of one or more complete blocks.

rule 1: (rule for deciding on counter_bits): The counter_bit sent with each message is sent so that for each $k$, $N(b_k) = N(s_k) + 1$, and $|b_k| = \max\{|s_k|, \log_2(N(b_k))\}$. 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. This is achieved by using a binary adder which is re-initiated at the initiation of each new block.

rule 2: (rule for deciding on colors): When receiving a message whose color is $c(b_k) \in \{0, 1, 2\}$, the new value of expected_color, which is the color $c(b_{k+1})$ of the next block $b_{k+1}$, is determined as follows: $c(b_{k+1}) = c(b_k) + zor_{N(s_k)+1}$ (mod 3). (This rule implies that $c(b_k) \neq c(b_{k+1})$, and that the sequence $(c(b_1), c(b_2), \ldots)$ is aperiodic if the sequence $(xor_{N(s_1)}, xor_{N(s_2)}, \ldots)$ is.) Computing $xor_{N(s_k)}$ is done by counting the counter_bits in $s_k$ (mod 2).

Note that both rule 1 and rule 2 above are easily implemented by a finite state machine.

4.1.2 Correctness and complexity proofs

Lemma 2: In any 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 equals expected_color. In any atomic step in which the sender does not receive such a message, it send a message, say $M$, whose color is expected_color. By the fairness of $E$, $M$ is eventually received, and the sender initiates a new block not later than when it receives $M$. \hfill \Box
A configuration in an execution is called a limit configuration if in the next step a new expected.color is computed; that is, the color of the next arriving message is equal to the present value of expected.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 queue (this block may be incomplete since upon receiving 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. We denote by Ck the limit configuration just before b_k is initiated.

Next we prove that the number of blocks in the limit configurations may only decrease.

Lemma 3: Let l_k be the number of blocks in the limit configuration C_k. Then l_k ≥ l_k+1, with equality only if s_k is a single block.

Proof: Let m_k ≥ 1 be the number of blocks in s_k. In the process of going from C_k to C_{k+1} one block was added (namely, b_k), and m_k blocks of s_k were removed, therefore we have that l_k+1 = l_k + 1 - m_k. The lemma follows.

Lemma 3 above implies that the number of blocks in the limit configurations can only decrease. Next we show that eventually this number must get down to one. First we need a technical Lemma:

Lemma 4:

(a) The sequence (xor_1, xor_2, ...) is aperiodic.

(b) Let (a_1, ⋯, a_i, ⋯) be an eventually periodic sequence. Then for each i, p > 0, the sequence (a_i, a_{i+p}, a_{i+2p}, ⋯, a_{i+mp}, ⋯) is also eventually periodic.

Lemma 5: Eventually, the number of blocks in the limit configurations is always one.

Proof: By Lemma 3 this number never increases, and hence it is eventually remains L for some constant L > 0 forever. We shall assume that L > 1 and derive a contradiction.

Call a limit configuration C_k ultimate if l_k, the number of blocks in C_k, is L. If C_k is ultimate then l_{k+1} = l_k and hence, by Lemma 3, s_k is a single block, which must be b_{k-L}. This means that the color of the block b_{k-L+1} must be equal to the color of the block b_k. In other words, c(b_{k-L+1}) = c(b_k), hence the sequence COLORS = (c(b_1), c(b_2), ⋯) is eventually periodic with period length L - 1 > 0. Let XOR = (xor_{N(b_1)}, xor_{N(b_2)}, ⋯). Then, c(b_{k+1}) = (xor(c(b_1)) + xor_{N(b_{k-L})} + 1) (mod 3). This means that the sequence XOR is also eventually periodic. We shall derive a contradiction by showing that the sequence XOR is aperiodic.

By Lemma 4 (b), in order to show that XOR is aperiodic, it is sufficient to show that for some positive i and p, the sequence XOR(i,p) = (xor_{N(b_i)}, xor_{N(b_{i+p})}, xor_{N(b_{i+2p})}, ⋯) is aperiodic. For this, observe that for an ultimate configuration C_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 XOR(i,L) = (xor_{N(b_i)}, xor_{N(b_i+1)}, xor_{N(b_i+2)}, ⋯), where N = N(b_i). Thus, XOR(i,L) is a suffix of (xor_1, xor_2, ⋯). This latter sequence is aperiodic by Lemma 4 (a), which implies that XOR(i,L) is also aperiodic. This yields the desired contradiction.

The complexity of protocol 3 is implied by the following lemma, whose proof is omitted.
Lemma 6: The number of messages in the links is non decreasing. Furthermore, for every large enough $k$, the number of messages in the limit configurations $C_k$ is $\lceil \log_2 |b_{k-1}| \rceil$.

5 concluding remarks

We have defined the new class of Self stabilizing message driven protocols. The viability of this definition was demonstrated by proving a non-trivial lower and upper bounds on the rate of growth of configuration size in any message driven protocol for token passing.

In the full paper we will present two application for the upper bound:

- Self stabilizing token passing protocols for any dynamic network.
- A general scheme for simulating any shared memory self stabilizing protocol by a message driven protocol.
References


