ELSEVIER Theoretical Computer Science 182 (1997) 159–170

Theoretical Computer Science

On the computational power of self-stabilizing systems [1]

James Abello [a,2], Shlomi Dolev [b,*]

[a] *Department of Computer Science, Texas A&M University, College Station, TX 77843, USA*
[b] *Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel*

Received July 1994; revised June 1995
Communicated by M.S. Paterson

Abstract

The computational power of self-stabilizing distributed systems is examined. Assuming availability of any number of processors, each with (small) constant size memory we show that *any* computable problem can be realized in a self-stabilizing fashion.

The result is derived by presenting a distributed system which tolerates transient faults and simulates the execution of a Turing machine. The total amount of memory required by the distributed system is equal to the memory used by the Turing machine (up to a constant factor).

1. Introduction

Our motivation to explore the power of interconnected processors with constant size memory was first triggered by the following questions: What is the relation between the computational power of a single powerful computer and a distributed system of limited power and memory processors that are subject to transient faults? The approach is different from the one taken by the parallel algorithm community [13]. The concern in this work is the fault tolerance of the algorithm rather than the time it takes to execute its task. We view a distributed system as a stand-alone system (as opposed to a single-site parallel machine that can be locally controlled) that runs on-going tasks and is able to overcome faults.

In particular, we are interested in *self-stabilizing* systems. A self-stabilizing system is a system that can be started in any *possible* global state. A *transient fault* is a fault that cause the state of a processor to change arbitrarily. Self-stabilizing systems can tolerate

* Corresponding author. E-mail: dolev@cs.bgu.ac.il. Part of this work was done while this author was at the department of computer science, Texas A&M University, College Station, TX 77843. Supported in part by TAMU Engineering Excellence funds and NSF Presidential Young Investigator Award CCR-9158478.

[1] An extended abstract of this paper was presented at the 6th International Conference on Computing and Information, Canada, May 1994.

[2] E-mail: abello@cs.tamu.edu. Supported in part by NSF grant CCR-9304081.

transient faults. When the intermediate period between two successive transient faults is long enough the system stabilizes. Following its stabilization the system demonstrates its desired predefined behavior.

In this paper we consider a distributed system of processors each with a constant amount of memory. In order to understand the inherent behavior of the system we examine the extreme case where each processor is equipped with only few bits of memory. The reference powerful computer is modeled by a Turing machine. Theoretically there is no upper bound on the amount of memory needed for storing the program that a computer needs to execute. This fact is also true in terms of Turing machines – where the program corresponds to the transition table. In order to eliminate the table size factor we consider only a specific deterministic universal Turing machine (denoted in the sequel by TM) [10].

Input and output are given in a distributed fashion. Each processor may receive part of the input, and should output part of the output. Since the processors have constant memory size, the input consists of no more than a constant number of bits. Note that the input length might be shorter than the number of processors in the system. To simplify presentation, when the number of processors is n and the input length is $l < n$, we concatenate the word $\perp^{n-l}$ to the original input word to obtain an input of length n. The output of the processors must (eventually) be correct with respect to the inputs. Note that the input can be changed during the execution of the algorithm. In this paper, we only focus on long enough periods of time in which the input is fixed and require that the output will correspond to the input some time after each such period begins.

Our distributed system is connected in a chain topology. The chain is only an abstraction of a predefined marked chain over any general graph. In particular, a system with a predefined ring (as is the case for token ring protocols) and a predefined leader fits this model. Each processor in the chain could be a communication port processor with limited resources of memory and computation. The number of processors in the chain is not a priori restricted. Obviously, any existing hardware is finite and the number of processors in the system is also finite. However, in order to have a base for comparison with the infinite tape of a Turing machine we do not a priori restrict the number of processors. For any given n we construct a self-stabilizing distributed system with n constant memory processors. A distributed system of n processors accepts (rejects) the input iff the corresponding Turing machine accepts (rejects, respectively) the same input using no more than n memory cells of its working tape. When the Turing machine uses more than n memory cells during the computation (or uses less memory cells but does not halt) the processors of the distributed system outputs, '$\perp$', as a "don't know" symbol.

Certainly, a TM can simulate the execution of any distributed system using the same amount of memory (up to a constant factor). Interestingly enough, the main result of this paper shows that processors with (small) constant amount of memory can tolerate transient faults and obtain the same result as a fault free execution of a TM. Namely, we show that a distributed system of interconnected constant-size memory processors can simulate the computation of a TM in the presence of transient faults. The total amount of memory required by the distributed system during the computation with the input word w is equal to the memory used by the TM with the input word w (up to a constant factor).

The study of self-stabilizing algorithms started with the fundamental paper of Dijkstra, [4], where three self-stabilizing algorithms for the mutual exclusion problem were presented. Recently, an extensive effort has been directed towards finding time and memory efficient self-stabilizing algorithms (cf. [1, 5, 6, 14]). Most recent works assume the existence of distinct identifiers (cf. [1, 5]). The use of distinct identifiers yields a lower bound of $\Omega(\log n)$ bits for the size of memory per processor. Thus, those solutions do not apply to systems with constant memory size processors. Other recent works use randomization in order to break symmetry (cf. [2, 8, 11, 12, 15]). In this paper we neither assume distinct identifiers nor use randomization. We restrict the system topology to be a directed chain with a *leader* processor at one endpoint and a *tail* processor in the other. Our system simulates a Turing machine computation in a self-stabilizing fashion.

In [9], it is proposed to use simulation in proving the possibility to force or preserve the self-stabilization property for different systems. The simulation of a TM by another system is not considered. In [16], the question of whether a polynomial self-stabilizing finite state program exists for decision problems is considered. Our goal is different, we simulate a Turing machine by a self-stabilizing distributed system of constant-memory processors in order to examine the computation power of the fault-tolerant distributed system. The remainder of the paper is organized as follows. In the next section we formalize the assumptions and requirements. Section 3 contains the description of our algorithm. Concluding remarks are in Section 4.

2. Distributed system

We consider distributed systems that consist of processors $P_1, P_2, \ldots, P_n$, that are connected in a chain. The processors are anonymous, the subscripts 1 to n are used only for convenience. No processor knows n the number of processors. Processors have sense of direction, i.e. for $j > 1$, P_{j-1} is the *left* neighbor of P_j and for $j < n$, P_{j+1} is the *right* neighbor of P_j. P_1 is the *leader* processor, P_n is the *tail* processor and the rest of the processors are *intermediate* processors.

The reader may refer to [6, 8] for the full (standard) definitions we use for configuration, atomic step, fair execution, and asynchronous round. Processors communicate by the use of shared communication registers.[3] Two neighboring processors P_i and P_j, communicate by two shared registers r_{ij} and r_{ji}. P_i (P_j) writes in r_{ij} (r_{ji}) and reads from r_{ji} (r_{ij}). In addition to accessing its neighbors communication registers, each processor P_i can repeatedly read one symbol of input from I_i, its *input register*, and

[3] In the context of self-stabilizing algorithms the use of shared communication simplifies the presentation with respect to the message passing model. However, our results can be applied to message passing systems as well by using methods presented in [7].

repeatedly write one symbol of output to its *output register* O_i. The content of I_i and O_i is either 0, 1 or $\perp$. We view the concatenation of the input symbols as a fixed word in $\{0,1\}^l \perp^{n-l}$ where $n \geqslant l$.

The *state* of a processor fully describes its internal state and value written in its registers including the output and input registers. A *configuration* is a vector of states of all processors. Processors execute *atomic steps*. An atomic step consist of some local computation followed by either a read from a communication register and input symbol or a write in a communication register and the output symbol. An *execution* of the system is a sequence of configurations $E = (c_1, c_2, \ldots)$ such that for $i = 1, 2, \ldots, c_{i+1}$ is reached from c_i by a single atomic step of some processor. Given an execution E, the first *round* of E is finished immediately after each processor has executed one atomic step; the second round is finished after each processor has executed one atomic step following the termination of the first round, and so on and so forth.

The requirements for self-stabilizing algorithms state the conditions under which the system has to stabilize when started in an arbitrary configuration and specifies the required behavior of the system following the stabilization period. Next we define the self-stabilization requirements for our distributed algorithm $\mathscr{A}$. Let w be a word in $\{0,1\}^l$. An algorithm $\mathscr{A}$ is self-stabilizing if for any finite n, when $\mathscr{A}$ is executed by a system of n processors and is started in *any possible* configuration, c, with input word $w \perp^{n-l}$ $(n \geqslant l)$ then: (1) any fair execution that starts with c has a suffix in which the output of every processor P_i is constant and, (2) this constant output is 1 (0, respectively), if the TM accepts (rejects) w using no more than n working tape cells, otherwise the output is $\perp$.

3. The reduction

A self-stabilizing mutual exclusion algorithm serves as a building block in our algorithm. The self-stabilizing mutual exclusion algorithm guarantees that starting with any possible configuration, after a finite number of asynchronous rounds every configuration contains exactly one processor which is executing the critical section. The mutual exclusion algorithm of [6] and the coloring algorithm of [8] use only constant number of states per processor and ensure that in every fair execution following the stabilization period the single token repeatedly "travels" from the leader to the tail and back. For simplicity we consider a processor that executes the critical section as holding a *token*. We use the terms *send* token and *receive* token to indicate transfer of the privilege to execute the critical section from one processor to another. Note that before a processor P transfers the privilege to execute the critical section, P can write in its shared communication register a "content" for the token. Thus, we view the token as an entity with a value that is transferred from one processor to another.

The (eventual) behavior of the token is used to ensure that the chain of processors will repeatedly write only the correct output. The processors repeatedly simulate the execution of the TM. Whenever a simulation of the TM computation is over, the processors are initialized to start a new simulation. The initialization does not effect the value of the output registers. The result of the simulation is overwritten to the output registers. Once the result of the simulation is correct any further write operation into the output registers (which is also a result of a correct simulation) does not change the value stored in these registers. A single step of the TM is simulated each time the token travels from the leader to the tail and back. Each processor contains the information of a single working tape cell. In every configuration of a correct simulation (one that follows correct initialization) there exists a single processor that is marked to hold the head of the TM. Whenever the token reaches the processor that holds the head of the TM the value of the working tape cell and the current state of the TM are used to calculate the transition of the TM. The transition includes modification of the contents of the working tape cell, change of the TM state and movement of the head mark to a neighboring processor.

Due to the self-stabilizing setting the simulation might not terminate. The following observation is used to ensure detection of a non-terminating simulation: A TM that reaches the same configuration twice in a single computation does not ever halt. Since our system is finite we propose to count the number of the TM configurations during the computation. To do so with a constant amount of memory per a processor we suggest using a distributed binary counter. Each processor maintains only two bits of the counter. The distributed counter is incremented by one in every step of the TM. The tail processor that holds the least significant bits starts to increment the counter whenever the token arrives to it. Indication of a carry is sent to the left neighbor when appropriate. If a counter overflow occurs before the TM accepts or rejects the input then the system is initialized.

In more detail, each processor P_i maintains two bits of the distributed counter in CntBits_i. When a token arrives to P_n, P_n computes the new value for CntBits_n and the carry. Then P_n writes the carry value in Tkn.Cr to its neighbor P_{n-1}. Whenever the leader P_1, detects a counter overflow the leader resets the system. The following activities occur during this reset (1) every processor P_i writes its input to WrkSym_i which is the i'th cell of the virtual TM working tape, (2) every processor P_i set a flag HdMrk_i to be false, the only exception is the leader (P_1) which sets HdMrk_1 to be true, (3) the binary counter bits of each processor are set to 00. Following the first reset the chain implements a virtual TM. The computation of the TM is simulated during the traversal of the token from the leader to the tail; when a processor P_i with $\text{HdMrk}_i = T$ receives a token that traverses in this direction the (constant space) TM table is used to determine the value for WrkSym_i, the movement of the head and the new TM state. Then the system reaches the new TM configuration by changing the virtual working tape, the head marker location and the TM state accordingly. Note that the token continues to the right. Hence, in case the direction of the head movement is towards the leader, the transition of the TM head is delayed until the token arrives from the direction of the tail.

Next we briefly describe the conventions, variables, functions and statements that are used in the code of the algorithm. Upon arrival of a token the code of a processor is executed sequentially from its beginning to its end; the labels that appear in the code (e.g. L1, L2) are used only for the sake of readability. The symbols '{' and '}' are used to denote the beginning and the end, respectively, of the portion of the code that is executed when the condition of the appropriate if statement is satisfied.

The token value: The token is a combination of three *field* values. Tkn.Cr: is used to indicate carry for the binary increment. Its value is either 0 or 1. Tkn.Rst: indicates a reset execution. Its value is either T or F. Tkn.TMSta: encodes the current state of (the universal) TM. Every possible state (of the constant number of states of TM) can be encoded in Tkn.TMSta. In addition $\perp$ is used to represent "no-state" during reset executions.

Local variables used by each processor: There are five local variables. RTkn: Stores the value of the token received. CntBits: This variable contains two bits of the distributed counter. HdMrk: Indication on the presence of the TM head. The value of HdMrk is eithr T or F. HdMov: Indicates the computed movement of the head. either Left, Right or Stay. WrkSym: The working tape symbol, every possible working symbol (of the constant number of working symbols of TM) can be encoded in WrkSym.

The functions used in the code: TM(Initial) – results with the initial state of TM. Tkn.TMSta, WrkSym$_i$, HdMov$_i$:= TM(Tkn.TMSta, WrkSym$_i$): uses the current Tkn. TMSta and WrkSym$_i$ and TM transition table to compute the next Tkn.TMSta, WrkSym$_i$ and HdMov$_i$. Note that we define the result of the statement TM($\perp$, WrkSym) to be the same as that of TM(TM(Initial), WrkSym).

The program of the leader P_1: Upon arrival of a token (from P_2) statements L1 to L6 of Fig. 1 are executed sequentially. L1: The value of the received token is stored in RTkn before it is modified. L2: This statement checks whether a counter overflow occurs, then checks whether the computation is over and initiates a reset if either happens. Following the examination for overflow, CntBits$_1$ are incremented by 1 (in case of overflow the result is 00). L3: This statement is executed when a head mark of TM is presented in P_1. In such a case the transition function is computed using the state of the TM as received with the token and the work tape symbol WrkSym$_1$. When the next head movement is towards P_2, P_1 clears the indication on the presence of the head and prepares Tkn.HdMrk to indicate on the transition of the head to P_2. P_1 assigns Tkn.Rst := T, when the next head movement causes failure of the head from the working tape. L4: When an indication on head movement from P_2 to P_1 is received, P_1 assigns HdMrk := T and clears the head transition indication in Tkn.HdMrk. L5: A reset indication (due to L2, L3 or arrival of a token with Tkn.Rst = T) triggers initialization of the work tape symbol, the head marker (at P_1), the counter bits and the Turing machine state. If the last computation has been terminated "normally" then the output symbol is the result of the computation and the Turing machine state is the initial state. Otherwise, the Turing machine state and the output symbol are set to $\perp$ to indicate "abnormal" termination. L6: P_1 sends the token to P_2.

```
ReceiveTail(Tkn)
L1: RTkn:=Tkn
L2: if Tkn.Cr=1 and CntBits$_1$=11 then
        {Tkn.Rst:=T}
    if Tkn.TMSta ∈ (accept,reject) then
        {Tkn.Rst:=T}
    CntBits$_1$:=CntBits$_1$ + 1
L3: if HdMrk$_1$=T then
        {Tkn.TMSta,WrkSym$_1$,HdMov$_1$:=
                TM(Tkn.TMSta,WrkSym$_1$)}
        if HdMov$_1$=Right then
            {Tkn.HdMrk:=T; HdMrk$_1$:= F}
        if HdMov$_1$=Left then
            {Tkn.Rst:=T}
L4: if RTkn.HdMrk = T then
        {HdMrk$_1$:=T; Tkn.HdMrk:= F}
L5: if Tkn.Rst=T then
        {WrkSym$_1$:=I$_1$; HdMrk$_1$:=T;
        CntBits$_1$:=00;
        if Tkn.TMSta ∈ (accept,reject) then
            {O$_i$:=TM(Tkn.TMSta);
            Tkn.TMSta:=TM(Initial)}
        else
            {O$_i$:=⊥; Tkn.TMSta:=⊥}}
L6: SendTail(Tkn)
```

Fig. 1. Algorithm for the leader P_1.

The program of the tail P_n: Upon arrival of a token (from P_{n-1}) statements T1 to T7 of Fig. 2 are sequentially executed. T1: The value of the received token is stored in RTkn before it is modified. T2: Similar to L3. The difference is in the case of head failure – a right movement implies head failure. T3: Similar to L4. T4: When a token arrives with a Tkn.Rst $= T$ the work tape symbol, head marker and counter bits are initialized. In addition the Tkn.Rst is assigned by F to indicate the completion of the reset. This last assignment is not executed when P_n initiates the reset (in T2). In such a case Tkn.Rst is sent to the leader which in turn resets the entire system. T5: The output is assigned according to Tkn.TMSta when the computation results with accept, reject or when a reset is initiated (i.e. Tkn.TMSta=$\perp$). T6: The counter is incremented by 1. T7: The token is sent to P_{n-1}.

The program of intermediate processor P_i: Upon arrival of a token from P_{i-1} statements I1 to I5 (Fig. 3) are executed sequentially. I1: The value of the received token is stored in RTkn before it is modified. I2: Similar to L3 and T2, with no head failure possibility. I3: Similar to L4 and T3. I4: A token with Tkn.Rst=T causes initialization of the working tape, head marker and counter bits. I5: The token is sent to P_{i+1}.

Upon arrival of a token from P_{i+1}, statements I6 to I11 (Fig. 4) are executed sequentially. I6: The value of the received token is stored in RTkn before it is modified. I7: The counter is incremented (when the token moves from the tail to the leader). I8: The head of the Turing machine moves from P_{i+1} to P_i. The assignment HdMov:=Stay

```
ReceiveLead(Tkn)
T1: RTkn:=Tkn
T2: if HdMrk$_n$=T then
        {Tkn.TMSta,WrkSym$_n$,HdMov$_n$:=
            TM(Tkn.TMSta,WrkSym$_n$)}
        if HdMov$_n$=Left then
          {Tkn.HdMrk:=T; HdMrk$_n$:= F}
        if HdMov$_n$=Right then
          {Tkn.Rst:=T; HdMrk$_n$:= F}
T3: if RTkn.HdMrk = T then
        {HdMrk$_n$:=T; Tkn.HdMrk:= F}
T4: if Tkn.Rst=T then
        {WrkSym$_n$:=I$_n$; HdMrk$_n$:=F;
        CntBits$_n$:=00}
        if RTkn.Rst=T then
          {Tkn.Rst:=F}
T5: if Tkn.TMSta ∈ (accept,reject,⊥) then
        {O$_i$:=TM(Tkn.TMSta)}
T6: if CntBits$_n$= 11 then
        {Tkn.Cr:= 1}
      else {Tkn.Cr:=0}
      CntBits$_n$:=CntBits$_n$ + 1
T7: SendLead(Tkn)
```

Fig. 2. Algorithm for the tail P_n.

```
ReceiveLead(Tkn)
I1: RTkn:=Tkn
I2: if HdMrk$_i$=T then
        {Tkn.TMSta,WrkSym$_i$,HdMov$_i$:=
            TM(Tkn.TMSta,WrkSym$_i$)}
        if HdMov$_i$=Right then
          {Tkn.HdMrk:=T; HdMrk$_i$:= F}
I3: if RTkn.HdMrk = T then
        {HdMrk$_i$:=T; Tkn.HdMrk:= F;
        HdMov$_i$:=Stay}
I4: if Tkn.Rst=T then
        {WrkSym$_i$:=I$_i$; HdMrk$_i$:=F;
        CntBits$_i$:=00}
I5: SendTail(Tkn)
```

Fig. 3. Algorithm for intermediate P_i, token towards tail.

```
ReceiveTail(Tkn)
I6: RTkn:=Tkn
I7: if Tkn.Cr=1 and CntBits$_i$=11 then
        {Tkn.Cr:=1}
      else {Tkn.Cr:=0}
      CntBits$_i$:=CntBits$_i$ + Tkn.Cr
I8: if RTkn.HdMrk = T then
        {HdMrk$_i$:=T; Tkn.HdMrk:= F;
        HdMov$_i$:=Stay}
I9: if HdMrk$_i$=T and HdMov$_i$=Left then
        {Tkn.HdMrk:=T; HdMrk$_i$:= F}
I10: if Tkn.TMSta ∈ (accept,reject,⊥) then
        {O$_i$:=TM(Tkn.TMSta)}
I11: SendLead(Tkn)
```

Fig. 4. Algorithm for intermediate P_i, token towards leader.

makes sure that I9 is not executed. I9: The transition of the head of the TM to P_{i-1} occurs when the token arrives from P_{i+1}. I10: Similar to T5. I11: The token is sent to P_{i-1}.

3.1. Correctness proof

The correctness hinges on the existence of a self-stabilizing mutual exclusion algorithm. In particular, the coloring algorithm of [8] guarantees that in any fair execution, after $O(n)$ rounds, a *safe configuration* for the mutual exclusion algorithm, c_{me}, is reached such that, in any configuration that appears after c_{me}, there exists at most one processor that executes the critical section. Moreover, following c_{me} the processors repeatedly execute the critical section in a fixed order, from the leader to the tail and back; at least one transfer of the token from a processor to its neighbor is made in every two successive rounds. Note that before the safe configuration c_{me} is reached there can be many tokens. In this period of time our algorithm does not operate correctly. Thus, when the mutual-exclusion algorithm stabilizes, the other part of the algorithm (that assumes the existence of a token that travels nicely from the leader to the tail and back) is in an arbitrary state. For example in such an arbitrary state more than one processor can have HdMrk$=T$. We prove that this part of the algorithm stabilizes too.

Lemma 3.1. *In every fair execution that starts with a safe configuration c_{me} of the mutual exclusion algorithm, the leader assigns Tkn.Rst:$=T$ at least once in every $4n2^{2n}$ rounds.*

Proof. Assume towards contradiction that the leader does not initiate a reset for $4n2^{2n}$ rounds. Let c_{l} be the configuration in the beginning of those $4n2^{2n}$. c_{l} follows c_{me} thus during those rounds the single token repeatedly travels from the leader to the tail and back, each such traversal takes no more than $4n$ rounds (i.e. two rounds for each move). In each traversal from the tail to the leader the binary counter is incremented by one. The value of the binary counter in c_{l} is an arbitrary non-negative number. Thus, if the leader does not initiate a reset during $4n2^{2n}$ steps following c_{me} then an overflow of the counter occurs and triggers a reset initialization. This contradiction proves the lemma. □

It is easy to see that following c_{me} whenever the leader sends a token with Tkn.Rst:$=T$ the token initializes the working tape bits, the counter bits, the place of the TM's head, and the TM state.

Lemma 3.2. *In any fair execution that starts with a safe configuration c_{me} of the mutual exclusion algorithm, after the leader assigns Tkn.Rst:$=T$ and sends the token, the token travels to the tail and every processor that receives the token initiates its variables.*

Let C be the number of states of a TM. Note that, by [10, pp. 173] and [17], $C \leqslant 56$. The next lemma is proved by a simple counting argument.

Lemma 3.3. *The number of different TM configurations with a tape of size n and $\{0,1\}$ alphabet is at most $Cn^2 2^n$.*

Proof. There are n possibilities for the place of the first $\perp$ in the working tape and at most 2^n possible working tape contents until the place of the first $\perp$. There are n possibilities for the location of the Turing machine head and C possibilities for the current TM state. □

Define a *reset initialization* configuration, c_{init}, to be a configuration that follows c_{me} such that c_{init} immediately follows an atomic step of the leader in which it assigns Tkn.Rst:=T. For every reset initialization configuration, c_{init}, we define a *reset termination* configuration, c_{term} to be the first configuration after c_{init} that follows an atomic step of the leader in which the leader receives the token.

Lemma 3.4. *In the second reset termination configuration after c_{me}, all the output registers are identical. Moreover the outputs are* 1 (0, *respectively*) *iff TM accepts* (*rejects, respectively*) *w with a working tape of size n. Otherwise, the output is* $\perp$.

Proof. First note that if a TM is in a certain machine configuration more than once before TM accepts or rejects then the TM does not halt. This observation is straightforward since the TM is deterministic and it repeats the same execution forever. Thus, if the execution encodes more than $Cn^2 2^n$ configurations and the TM does not accept or reject w then the TM reached a certain configuration at least twice and it will never halt. By Lemma 3.2 and by the algorithm following the first reset termination configuration that follows c_{me} the distributed system simulates the TM computation and counts the TM configurations. Thus, a reset is triggered when either (1) the TM reaches a state in which it accepts or rejects w or (2) when the counter reaches $2^{2n} > Cn^2 2^n$ (for $n > 14$) or (3) when the head of the TM attempts to move to the left of the leader or to the right of the tail. Hence, before the second reset termination configuration the outputs of the processors are correctly set. □

Theorem 3.5. *In every fair execution after* $O(n2^{2n})$ *rounds, in every configuration the output of each processor is accept* (*reject*) *if the TM accepts* (*rejects*) *w with working tape of size n. Otherwise, the output is* $\perp$.

Proof. By Lemma 2 of [8], c_{em} is reached within $O(n)$ rounds. By Lemma 3.1, the first c_{term} appears during the first $4n2^{2n}$ rounds that follows c_{me}. By Lemma 3.4 during an additional $4n2^{2n}$ rounds each processor either receives a token with TMSta that indicates acceptance or rejection or with Tkn.TMSta=$\perp$. Since any further computation that begins with a later reset is identical, the output is not changed. □

4. Concluding remarks

In this paper we investigated the computational power of self-stabilizing systems with constant memory size processors. Interestingly, interconnected processors with a (small) constant amount of memory can tolerate transient faults and obtain the same result as a fault free execution of a TM. This implies that when there is an embedded ring with a leader in a system with constant memory size processors, the system copes with transient faults and still has the computational power of a TM with the same total amount of memory (up to a constant factor).

The algorithm presented in Section 3 requires $O(n2^{2n})$ rounds to stabilize. The algorithm can be accelerated by the use of an upper bound on the execution time of the TM. When an upper bound $t(w)$ on the execution time of the TM (for an input word w) is known, and could be given as a part of the input, then our algorithm can stabilize within $O(nt(w))$ time. The use of a counter to reinitiate the system by counting the steps in the execution and comparing with a given bound had been previously used in, e.g., [3, 9, 16]. To accelerate the algorithm, we suggest implementing the counter in a distributed fashion using constant memory per processor. The modified algorithm will compare the input step bound with the step counter and give an indication to the leader when the step counter exceeds the step bound.

Acknowledgements

We would like to thank the editor Mike Paterson and the anonymous referees for their constructive comments. We also thank Alex Ribman for simulating the code.

References

[1] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir and G. Varghese, Time optimal self-stabilizing synchronization, *Proc. 25th Annual ACM Symp. on Theory of Computing* (1993) 652–661.

[2] B. Awerbuch and R. Ostrovsky, Memory-efficient and self-stabilizing network reset, *Proc. 13th Annual ACM Symp. on Principles of Distributed Computing* (1994) 254–263.

[3] B. Awerbuch and G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols, *Proc. 32nd IEEE Symp. on Foundations of Computer Science* (1991) 258–267.

[4] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, *Comm. ACM* **17** (11) (1974) 643–644.

[5] S. Dolev, Optimal time self-stabilization in dynamic systems, *Proc. 7th Internat. Workshop on Distributed Algorithms* (1993) 160–173.

[6] S. Dolev, A. Israeli and S. Moran, Self stabilization of dynamic systems assuming only read/write atomicity, *Proc. Ninth Annual ACM Symp. on Principles of Distributed Computation*, Montreal (1990) 103–117; *Distrib. Comput.* **7** (1993) 3–16.

[7] S. Dolev, A. Israeli and S. Moran, Resource bounds for self-stabilizing message driven protocols, *Proc. Tenth Annual ACM Symp. on Principles of Distributed Computation*, Montreal (1991) 281–294; a journal version to appear in *SIAM J. Comput.*

[8] S. Dolev, A. Israeli and S. Moran, Uniform dynamic self-stabilizing leader election, *Proc. 5th Internat. Workshop on Distributed Algorithms*, Delphi (1991) 163–180; Part of the results appears in *IEEE Trans. Software Eng.* **21** (5) (1995) 429–439 and TR 94-039, Dept. of Computer Science, Texas A&M Univ.

[9] M.G. Gouda, R.R. Howell and L.E. Rosier, The instability of self-stabilization, *Acta Inform.* **27** (1990) 697–724.

[10] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA, 1979).

[11] A. Israeli and M. Jalfon, Token management schemes and random walks yield self stabilizing mutual exclusion, *Proc. Ninth Annual ACM Symp. on Principles of Distributed Computation*, Montreal (1990) 119–130.

[12] G. Itkis and L. Levin, Fast and lean self-stabilizing asynchronous protocol, *Proc. 36th Annual IEEE Symp. on Foundations of Computer Science* (1994) 226–239.

[13] R. Karp and V. Ramachandran, A survey of parallel algorithms for shared memory machines, Technical Report UCB/CSD 88/408, Computer Science Division, Univ. California, 1988; also in *Handbook of Theoretical Computer Science*, J. van Leeuwen, ed. (Elsevier, Amsterdam, 1990) 869–941.
[14] L. Lamport, The mutual exclusion problem: Part II – Statement and solutions, *J. ACM* **33** (1986) 327–348.
[15] A. Mayer, Y. Ofek, R. Ostrovsky and M. Yung, Self-stabilizing symmetry breaking in constant-space, *Proc. 24th ACM Conf. on Theory of Computing* (1992) 667–678.
[16] M. Schneider, Self-stabilizing real-time decision systems, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems* (Kluwer Academic, Dordrecht, 1995).
[17] C.E. Shannon, A universal Turing machine with two internal states, *Automata Studies* (Princeton Univ. Press, Princeton, NJ, 1956) 157–167.