Chapter 1

Self-Stabilizing Algorithms for Synchronous Unidirectional Rings*

Alain Mayer† Rafail Ostrovsky‡ Moti Yung§

Abstract

In this work we investigate the notion of built-in fault-tolerant (i.e. *self-stabilizing*) computations on a *synchronous uniform unidirectional* ring network. Our main result is a **protocol-compiler** that transforms any self-stabilizing protocol P for a (synchronous or asynchronous) **bidirectional** ring to a self-stabilizing protocol P' which runs on the synchronous **unidirectional** ring. P' requires $O(S_{LE}(n)+S(n))$ space and has expected stabilization time $O(T_{LE}(n) + n^2 + nT(n))$, where $S(n)$ $(T(n))$ is the space (time) performance of P and $S_{LE}(n)$ $(T_{LE}(n))$ is the space (time) performance of a self-stabilizing leader-election protocol on a bidirectional ring. As subroutines, we also solve the problems of leader election and round-robin token management in our setting.

1 Introduction

The design of efficient distributed algorithms for *unidirectional* networks has proven to be a difficult task. There are only a few known protocols, e.g., [15, 31, 2, 25, 16, 28] and most of them do not consider the issue of *fault-tolerance* or that of symmetry breaking (they assume a given leader). Here, we are taking the first step in investigating simultaneously strong fault-tolerance (self-stabilization) and uniformity.

The notion of *self-stabilizing* protocols, suggested as early as [12], has been extensively studied in recent years. Such protocols have a very strong fault-tolerance property: starting from an arbitrary state they automatically recover into (and further maintain) a legal state. Most of the work in self-stabilization has dealt with bidirectional networks, e.g. [20, 3, 13, 24, 18, 34, 9, 10, 7, 5, 27, 1, 35, 30, 8, 23]. Many of the techniques and algorithms developed in the above papers rely heavily on repeated communication of each node with all of its neighbors using bidirectional links. Examples include: random walks (allowing move of tokens to any neighbor), local checking where neighbors exchange state information, a hand-shake between neighbors, and other bidirectional data-link mechanisms.

Motivation:
In this work we study self-stabilizing protocols for uniform unidirectional rings. Many basic issues in distributed computing arise from restricted communication (locality), faults and uniformity. In particular, rings are the basic ground for investigating the fundamentals of symmetry in distributed computing ([6, 21, 17, 31, 15, 11, 26, 22]) to mention just a few examples).

From a practical point of view, various flavors of token rings (see [33]) are popular examples of unidirectional ring structures. Our network model is motivated by the FDDI (Fiber Distributed Data Interface), a high-performance 100 Mbps fiber optic token ring (see [33] pages 166-168, [32], [4]).

The FDDI architecture, (which has available implementations and which is ANSI X3T9.5 standard), is centered around two unidirectional counter-rotating fiber-optic rings which operate independently. A station may be connected to both or just to one ring. The FDDI is synchronous (clocks are required to be stable up to 0.005 percent, assuring minimal drift and preventing buffer overflows). As part of the protocol, stations count time and may time out if control signals do not propagate fast enough. Messages include control bits for uniform processing at a low layer of the architecture.

Much care has been taken to assure fault tolerance in FDDI. Nodes must have optical bypasses, so that in case of failures the network operates. Also, the low layers of Physical and MAC (Media Access Control) have fault tolerance mechanisms for recovery of tokens and other control messages. The processings at that level is via finite state machines. Each node has an "elasticity buffer" which is used to receive messages into and send messages from. Concurrent transmissions are allowed and are synchronized (we employ constant size messages and their concurrent transmissions in our model). We note that in a unidirectional archi-

*Extended summary

†Columbia University. E-mail: mayer@cs.columbia.edu.

‡Bell Communications Research, MCC 1C-341R, 445 South Street, Morristown, New Jersey 07960-6438, E-mail: rafail@bellcore.com.

§IBM Research, T.J. Watson Research Center. E-mail: moti@watson.ibm.com.

tectures with limited intermediate buffers at the nodes, it makes sense to operate in a synchronous mode since otherwise frequent message losses (due to overwriting message buffers) may occur and reduce the network utilization. Our results can be viewed in light of the FDDI architectural model, as unified control algorithms employing small space and short time and possessing a built-in efficient fault tolerance capabilities which are superior to the standard fault tolerance mechanisms in time, space, and uniformity (see, for comparison, the recovery protocols described in [4], e.g. the token recovery, re-synchronization "Beacon" procedure, and election procedures).

Our Results:
In this work we provide a general tool for designing self-stabilizing protocols, by way of reduction. Namely, we give a (space and time efficient) *protocol compiler* which translates any self-stabilizing protocol for a bidirectional ring into a self-stabilizing protocol running on a uniform, synchronous, unidirectional ring. Hence we enable the protocol-designer to study the difficult aspects of fault-tolerance on the conceptually simpler mode of a bidirectional ring.

While developing the general solution, we also give specific and more efficient solutions to the tasks of *token management* (assuring that one token is circulating in a round-robin fashion in the ring) and to *leader election* (which assures that one and only one processor gets marked as a leader). Both procedures are basic tasks on ring networks; both are crucial parts in the general compiler.

To obtain the above results, we reduce from self-stabilizing leader-election on a bidirectional ring (with $S_{LE}(n)$ and $T_{LE}(n)$ being its space and stabilization time) as a black box.

Another procedure we need is an efficient simulation of a bidirectional round of concurrent communication when each processor on the ring sends messages to both its neighbors. Given a leader, we show how to achieve this global task optimally in time and space. We employ a reduction from the "Firing Squad" problem (see, for example, [28]).

Combining these techniques, we get as our main result: A compiler which transforms any self-stabilizing bidirectional protocol P with space $S(n)$ and stabilization time and $T(n)$ into a unidirectional protocol P' with $O(S_{LE}(n) + S(n))$ space and expected stabilization time of $O(T_{LE}(n) + n^2 + nT(n))$. (Hence, using the results of [23], any protocol for *bi-directional* ring which takes $O(1)$ space per processor (i.e. ring of automata) and polynomial stabilization time can be run with $O(1)$ space (i.e., by automata) and polynomial stabilization time on a *unidirectional* ring as well.)

Our solution, in fact, shows that in our target architecture, the limited access to neighbors does not reduce the set of problems that have a self-stabilizing solution (as one could have assumed based on the developments and techniques which were available till now). Furthermore, performance is not impaired by much since the penalty in space is minimal and since our model requires $\Omega(n)$ time units just to simulate the simple procedure of passing a single message to a direct neighbor. (After stabilization, the overhead is $O(1)$ in space and $O(n)$ in time).

2 The Model

We consider *unidirectional* rings of uniform processors, (the most restrictive topology from symmetry and connectivity perspectives). The processors are uniform since they run exactly the same code and are not accessing their unique IDs. Processors communicate by synchronous message passing, a node can send a message to its (unique downstream) neighbor and at the start of the next step that neighbor can read this message from an input buffer. The processor has an input register, an output register and a state register. It has a unique message buffer. It can read the (control and data) information at each time unit off the message buffer and process it.

We remark that in all our algorithms, it will read a constant portion off the message to decide what to compute (our control information is constant), and the data portion is used as a "black-box" left by the control to the node application algorithm.

We consider randomized solutions (otherwise, token management and leader election– and thus other general tasks, are impossible as was already shown in [12]). Thus, each processor can flip (uniformly random) independent coins at any computation step. We concentrate on small-memory solutions and if a processor needs to remember an outcome of the coin-flip we require additional registers for this. (Note that with small memory processing typical in hardware processing at a switch/coupler, it makes good sense that processors do not access their unique ID's which size is of course logarithmic in the network size). The ring is *synchronous*: all processors of the network simultaneously agree on an identical clock-tick signals and the computation is naturally divided into time-steps, according to the clock-ticks. In each clock tick the processor, based on its local state and the message received from its neighbor, makes a transition into a new state and may generate and send a message to its unique neighbor (which will receive it at the next clock tick). Maintaining tight clock synchronization is typical in local area networks (e.g. FDDI).

The fault model assumes that any processor's memory can be put into an arbitrary initial state, while the processor's program code is protected (e.g. hard-wired).

This is the model for self-stabilizing protocols [12], which captures the issue of automatic recovery from spurious memory initializations, transitions and changes. (Parenthetically, we remark that strengthening the fault model to include program code failure in distributed environments is possible but is more complicated and requires redundancy and other constraints [29]).

The *global state* of the system is defined as the cross-product of the states of the processors and the contents of the message buffers on links. A set of *legal* global states is defined for each problem.

An algorithm is said to be *self-stabilizing* if starting in an arbitrary global state (which models the network's state after the initial memory faults and topology changes), the algorithm guarantees that (i) eventually a legal state is reached and (ii) every further state is legal.

For example for the problem of token management, the goal is to have a single token circulating in the ring in a round-robin fashion (for network control). The initial state of the ring may contain many tokens or zero tokens. Naturally, a global state in which there is exactly one token (in a message buffer) at a node, and the state of this node correctly reflects that the token was last passed to that node, and will be next passed to its (active) neighbor is a legal global state.

3 Problems Considered and a Road-map

Our general protocol is constructed by a chain of reductions among building block protocols.

First we show how to generate a "binary clock", by having every node agree to call each clock pulse a zero-pulse or a one-pulse. Disagreement in the initial state is resolved via coin-flips (randomization). This protocol stabilizes in expected $O(n^2)$ time.

Using the binary clock we consider how to simulate a moving ring on top of the stationary ring as follows: On zero-pulses messages are transmitted downstream and on one-pulses the messages are left behind and the state of the nodes is transmitted downstream. Thus we have a bidirectional ring in motion with respect to the stationary unidirectional ring. We then can use a self-stabilizing leader election protocol on the bidirectional ring to obtain a unique moving leader. This leader is equivalent to a rotating token.

<u>TOKEN MANAGEMENT</u>:
The goal of a token management scheme is to obtain a unique token circulating around the ring. We show that:

Theorem 3.1 There exists a token management scheme for unidirectional rings with space $O(S_{LE}(n))$ and expected stabilization time $O(T_{LE}(n)+n^2)$.

Our next step is to reduce stationary leader election from "moving ring leader election": By operating on two bits on the token the nodes elect a stationary leader. One bit is used to detect the absence of a leader, the other to detect the presence of multiple leaders.

<u>LEADER ELECTION</u>:
Every processor has a "leader"-bit. If a processor has set this bit, we say that it is chosen as a "leader". In a self-stabilizing leader election protocol, regardless of the initial state, after a given (stabilization) time, exactly one processor has set its leader-bit.

Theorem 3.2 There exists a leader election protocol for unidirectional rings, with space $O(S_{LE}(n))$ and expected stabilization time $O(T_{LE}(n)+n^2)$.

<u>COMMUNICATION ROUND SIMULATION</u>
We assume that a leader is given. Given a communication round in a bidirectional protocol over a ring of fixed size buffers (which may be defined asynchronously), we can efficiently simulates this round. In particular, we are interested in the concurrent message transmission of $2n$ messages (namely when each node sends a message to its downstream and upstream neighbors in the protocol). For the concurrent transmission or when the bidirectional algorithm requires that in each round some processor sends a message to its upstream neighbor, our simulation is time optimal. Our simulation is also space optimal. The problem is reduced from firing squad which generates a global synchrony in message passing on the ring.

Theorem 3.3 There is a message transmission protocol whereby a round of message transmission in a bidirectional ring protocol, can be simulated on a unidirectional ring with an additional $O(1)$ space and in $O(n)$ time units (in fact smaller than $5n$). Further, the concurrent message transmission problem (where each processor sends to both its neighbors) is in $\Omega(1)$-space and $\Omega(n)$-time on unidirectional ring with a leader.

The above theorem, in fact, is a general efficient simulation (perhaps of an independent interest). It also implies the next result. Using the stabilized leader and the round simulation, we show how to efficiently simulate a bidirectional protocol on our unidirectional rings by employing the leader and a linear-time constant space "firing-squad protocol". (Note that the original simulated bidirectional protocol can be synchronous or asynchronous).

Bidirectional Ring Simulation:
We assume that a self-stabilizing protocol for a bidirectional ring is given. Also given are the above procedures. We show how to simulate the bidirectional algorithm on a unidirectional ring maintaining self-stabilization. By simulation we mean that there is an algorithm that uses the variables of the bidirectional algorithm, and auxiliary control state and message variables that are part of the compiler that "runs" the bidirectional protocol. The compiler, using the auxiliary control, "simulates" the steps of the bidirectional algorithm on the unidirectional architecture.

More formally, given a bidirectional algorithm there is a definition of global legal state in the bidirectional algorithm and transition function (that by the correctness of the bidirectional algorithm moves after stabilization from one legal state to another). Thus, in the simulation, there is a unidirectional phase that will emulate any bidirectional transition (in the sense that it affects identically the same processors and variables that are affected by the transition in the bidirectional algorithm), and after expected finite time (small polynomial in our case) the transition will be completed. Also, there is a definition of a global legal state of the simulation that will correspond to a legal state of the bidirectional algorithm when taking the simulation global state and projecting it over the variables that correspond to the bidirectional algorithm. We give this (main) result:

Theorem 3.4 There exists a protocol-compiler that transforms any self-stabilizing protocol P for a bidirectional ring into a self-stabilizing protocol P' which runs on a synchronous, uniform, unidirectional ring. P' requires $O(S_{LE}(n) + S(n))$ space and has expected stabilization time $O(T_{LE}(n) + n^2 + nT(n))$, where $S(n)$ ($T(n)$) is the stabilizing space (time) of P. After stabilization, P' takes additional $O(S(n))$ space, and $O(nT(n))$ time to perform its (sub)-task(s).

4 The Basic Protocols

We show the proof of our main theorem in four steps. In the first three steps we show a leader election protocol on a ring, and then show how given a self-stabilizing leader election, a simulation of bidirectional communication can be carried out in a self-stabilizing fashion.

4.1 Binary clock

Recall that our setting is synchronous. That is, all processors simultaneously hear clock-ticks. However, all the clock-ticks are "identical". That is, a clock-tick that every processor receives can be interpreted as a "unary" message, say a 1. In our solution, we need a larger degree of synchrony. Towards this goal, we introduce a *binary clock* defined as follows: binary clock is a clock which emits an alternating $0, 1$ sequence, and all processors simultaneously receive this sequence (i.e. there is never a case where one processor receives a 0 while some other processor receives a 1,, etc. It is always the case the all processors simultaneously receive a 0, then a 1, then a 0, etc.). We stress that we do not have a binary clock, but rather, show how our synchronous system (i.e. the system with a "unary" clock) can self-stabilize to a "binary clock".

It is easy to see that given proper initialization, "binary clock" is easy to implement. Every processor keeps a single bit variable B, which is initialized to zero at each processor. With each clock-tick, every processor toggles the bit. Since all the processors are in agreement when their bit is zero and when it is a one, this constitutes a good binary clock.

In case when there is no proper initialization, processors might be in disagreement about the value of the binary clock. Our objective is to make then all agree which unary clock-tick corresponds to zero and which to one. Below we describe the algorithm that performs this task.

Self-stabilizing binary clock algorithm:

Our algorithm is as follows: every processor keeps a single bit variable which indicates whether the unary clock (the tick) should be interpreted as a 0 or as a 1 of a binary clock. At each clock-tick every processor toggles its binary-clock variable. In addition, every processor sends its variable value at each step to its (unique downstream) neighbor. If a processor is in disagreement with the binary clock of its up-stream neighbor, it flips a coin, and with probability $\frac{1}{2}$ does not toggle its variable in the next step (i.e. sends the same bit twice), and with probability $\frac{1}{2}$ toggles its bit as usual:

Protocol on each node for "binary clock"
state:
boolean B; /* value representing the clock */
boolean U; /* clock from up-stream neighbor */
boolean α; /* outcome of the coin-flip */

receive U from up-stream neighbor;
set α := coin-flip;
if ($U = B$ or $\alpha = 0$) **then**
  $B := 1 - B$;
  send (new) B to its down-stream neighbor;
else
  send (old) B to its down-stream neighbor

Lemma 4.1 The binary clock algorithm stabilizes to a binary clock in expected $O(n^2)$ steps.

Proof:
First, note that if all processors are in agreement, they will never get out of the agreement.

Suppose processors are not in agreement. Divide the processors into segments, where in each segment processors are in agreement. Consider boundaries of this segments (their active dynamic head processor). Clearly, there are even number of them (as the segments form a two-coloring on a cycle of length equals number of segments, thus it must be even), and they move (downstream) with probability half. Moreover, if two boundaries meet, they disappear (i.e., they meet by chewing one segment and the two segments around it unite into one segments– reducing the total number of boundaries by two). Thus, we can reduce the binary-clock stabilization to the following combinatorial problem:

There is a ring of n processors, on which even number of "tokens" (representing borders) are placed. At each step each token moves in the same (say, clockwise) direction with probability half. If two tokens meet, they disappear. We are interested in the expected number of steps before all tokens disappear.

Standard random walk analysis, similar to Israeli and Jalfon [20], gives $O(n^2)$ expected time before all the tokens disappear. ∎

4.2 Simulation of a Bidirectional "Moving Ring"

Our next simple step is to show how any protocol for a bidirectional ring can be executed on a "virtual bidirectional ring" which moves *on top* of the unidirectional ring. This natural idea is known in the folklore and was used implicitly earlier. Nevertheless, it is useful

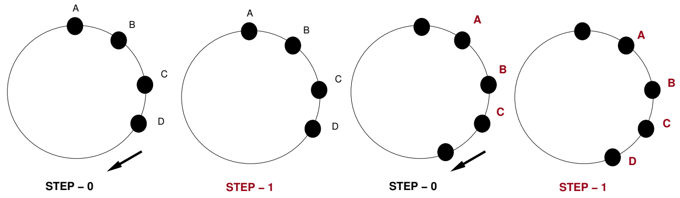

Figure 1: Merry-go-round algorithm

to point this out explicitly, and claim that it is usable even in our context of self-stabilization where the moving ring has to move forever! (as we will see).

The Merry-Go-Round (Carrousel) Algorithm:
Consider a ring of "virtual" processors which sits on top of the unidirectional ring. The "virtual" ring of processors is moving at half-speed: at each zero-pulse of the binary clock it stays in its place, at one-pulse the entire ring moves one-step downstream (in relation to the stationary ring of processors). Stationary processors simulate the computation of moving processors and when the ring moves they send the description of the finite-state-control of the processor to their respective downstream neighbors.

The moving ring can simulate the bidirectional communication: "virtual processors" on a moving ring can send messages down-stream when the ring is stationary (Step-0 in Figure 1), and can send messages to their upstream neighbors by leaving the messages "behind" when the ring moves (Step-1 in Figure 1).

In the description below, we show how an arbitrary protocol P for a bidirectional ring can be executed on the "moving ring".

Protocol on each node to simulate P on a "moving ring":
state: S, S'; /* state-descriptor of P */
left-behind-msg; /* msg for upstr neigh */
boolean B; /* binary clock */
if $B = 0$ (stationary) **then**
  Execute current step of P according to S;
  left-behind-msg := current upstream message;
if $B = 1$ (moving) **then**
  Send S to (downstream) neighbor;
  receive S' from upstream neighbor;
  set current msg from downstream (in S')
    to left-behind-msg;

Lemma 4.2 Any self-stabilizing protocol P for a bidirectional ring is synchronously and correctly executed on the "moving ring" with a slowdown factor 2.

4.3 Leader Election

Given the above simulation of a "moving ring", we can execute a self-stabilizing leader election protocol (such as described in [8, 23, 27]) for bidirectional rings on the "moving ring". Hence, we obtain a *"moving leader"* on the "moving ring" (or, equivalently, a unique circulating *token* on the unidirectional ring). Thus, in fact, this constitutes the proof of Theorem 3.1.

Next, we show how this can be used to get a stationary leader on the unidirectional ring, which completes the proof of Theorem 3.2.

Stationary Leader Algorithm:

In the following we show how to elect, in a self-stabilizing fashion, a leader among the nodes in the ring, given that we have a unique, circulating token (i.e., a moving leader).

The control-messages for our algorithm is just the token which carries two additional bits; the L- and M-bit. Also, every node has these two additional bits as part of its state. The L-mechanism ensures that the ring stabilizes to at Least one leader. The idea for the L-mechanism is that the nearest upstream (with respect to the token-movement) leader always guarantees that a non-leader does not see the same L-bit twice in a row on the passing token. That is, if a leader sees the token with the same L-bit as its own L-value, it will flip the bit on the token. If a non-leader sees the token with the same L-bit as itself (i.e., supposedly, it has the same value as before) it will turn into a leader (and also flip the L-bit on the token). The M-mechanism ensures that the ring stabilizes to at Most one leader. The idea of the M-mechanism is that if a leader sees a different M-bit on the token than it has itself, it assumes that there is more than one leader and hence turns into a non-leader; otherwise it will flip a coin and store the outcome as its (and the token's) new M-bit. This is the continued checking for non-uniqueness of the leader.

Protocol for a node receiving a token T:
if $\neg$leader **then**
  if $L_T = L$ **then**
    $L_T := \neg L_T$; leader := true; $M := M_T$
else {leader }
  if $L_T = L$ **then** $L_T := \neg L_T$;
$L := L_T$;
if leader **then**
  if $M_T \neq M$ **then** leader := false
  else M_T := coin-flip; $M := M_T$

Lemma 4.3 Given a unique token, the (constant space) algorithm stabilizes to a unique leader in expected $O(n \log n)$ steps.

Proof: We first define a predicate *cover* over the global state of the ring as follows: cover(R) = ($\forall i : L_i = L_{i-1} \vee$ *leader*$_i$ $\vee$ token between $i-1$ and i, s.t. $L_T = L_{i-1}) \wedge M_T = M_j$ (where j is closest upstream leader w.r.t. the token).

First note that cover(R) holds after one round of the token: Every node receiving T will copy its L-value and only if it is a leader it will change L_T before copying and every leader copies the value of M_T.

So from now on we assume that cover(R) holds. A node turns into a leader iff there is no leader: if there is no leader then within one round $L = L_T$ must hold at some node since only a leader changes the value of L_T; if there is at least one leader then for every non-leader i there is one upstream leader, such that when the token passes through that leader and it has the same L-value as i it will flip its value. A leader is removed only if there is more than one leader and in every round of the token the expected number of leaders is divided by two: The token always carries the M-value of the leader it saw last and thus a mismatch can only occur if there are at least two leaders. Every time a token arrives at a leader l and did not eliminate the last leader it passed through, it carries an M-value which is random and independent of l's M-value, which implies the expected time. ∎

We can employ (for our simulation) concrete bidirectional implementations of leader election protocols. They are readily available and have various flavors (some are more practical and some are more theoretical but economical in one measure or another). For example, combining [27] and [8] we can get leader election on unidirectional rings which takes time $O(n \log^2 n)$ and space $O(\log^* n)$. (This is implied by the fact that [27] assumes constant space and takes $O(n \log n)$ time assuming a token exists, and [8] takes $O(n \log^2 n)$ and space $O(\log^* n)$ and can reset the computation by introducing a token in case no token exist; further [27] transforms this into a bidirectional leader election within the same time and space). A more space efficient procedure that takes constant space to elect a leader bidirectionally can be performed using the constant space computations of [23] (which will result in a constant space protocol after compilation, thus we can implement it as a cellular automata algorithm).

5 The General Simulation of a Bidirectional Ring

Next we give the general simulation. We show first how to simulate a round of communication in which each processor sends a message to its two neighbors when a unique leader is present. We assume all processors send a message (which may be the empty one).

The round simulation is presented in two stages: first asynchronous mode, which demonstrates how in the case of such mode of timing the message passing should be carefully managed (to avoid overwriting of messages in the single buffers at the nodes). Then, the second stage is a synchronous simulation, which demonstrates that with synchrony we have a larger bandwidth since concurrent safe transmissions are assured.

Once we can simulate a round, we will proceed and put all the subroutines together, showing how to perform a general self-stabilizing protocol compiler on uniform rings.

5.1 Asynchronous communication round simulation

Given a leader node the leader sends a "start-the next-round" message around, declaring the start of the next round. With this message each node also sends its forward message. Next, we will serve the ring for $O(n^2)$ steps simulating the sending of backward messages to complete a single message sending round of the bidirectional ring. The leader node sends a token n times around. In the first round it marks itself as a *receiver* and the node next to it gets the token marked as *sender*, it takes the marker off the token and appends to it the message instead. Once the receiver gets the (possibly empty) message, it reads the message and forwards the token with the *receiver* marker forward to the current sender, the current sender now forwards the token with the *sender* marker to its neighbor which becomes the new sender and a message is sent from the sender to the receiver going around the ring. This process continues untill the leader becomes the *receiver* again, it first sends a "start-the next-round" message with the token that goes around (which also takes care of forward messages) and the round continues.

5.2 Synchronous communication round simulation

Next we give our method which is based on constant-space firing-squad implementation on a synchronous unidirectional ring and the time over-head is $O(n)$ which is optimal for certain global problems [19] (the firing squad procedure is also described in [16, 28]). We note that similar marking techniques have been employed in [2] for the problem of marking the upstream neighbor (but not for firing squad or for concurrent message transmissions).

Lemma 5.1 At the end of the protocol for round communication any processor holds its both (upstream and downstream) neighbors' messages. The protocol requires $O(1)$ space and $5n$ time units for concurrent transmission of all round messages.

Proof: (sketch)

The idea is that we can exploit a firing squad protocol that can be started by the leader. This protocol ends up when all the nodes fire at the same time.

In fact the following steps are taken place

1. The leader sends a special token marked *start-2* to move on the moving ring (circulating one step forward each time). (If this is the starting round, extra marking will be needed as will be explained).

2. One step later the leader starts a "firing squad" procedure.

3. When the firing squad of step 2 fires, every processor sends a message to its immediate neighbor (which is received) and a message to be sent backwards which is put on the moving ring (i.e., to circulate).

4. The task is now to stop the moving ring one-step before its starting position; this assures us that each message stops one step ahead of its origin, namely at its destination. This is done by the leader activating the firing squad exactly when the special token *start-2* reaches it.

5. When the second firing squad fires, each node receives the circulating message (off the moving ring).

Note that when the first firing squad started the special token was at the neighbor one-step ahead of the leader, at the moment of the first firing messages are put on the moving ring and the special token is at processor number x. The second firing squad starts when the special token is at the leader itself (one step backwards). The time from activation till firing is the same in both firing squad activations (it is a deterministic procedure). Thus, the special token ends at the second moment of firing at processor number $x-1$, which is the destination of the message of processor x. Since all messages are moving together (on the moving ring), we conclude that for any message sent by any processor number y, that message ends up at processor $y-1$ (i.e. at its destination).

∎

5.3 The general compiler

We simulated a round of message transmission using linear time. Theorem 3.4 can now be proved, putting all the previous results together.

We assume that each processor keeps an initial-state and a current-state. We assume that a leader when just elected, has a bit "just-elected-leader" turned on; when this bit is true it causes the leader to re-activate the compiler. It does so by starting the first round of simulation and by marking the special *start-2* token with a special marking of "restart", when the token gets back to the leader first time, it removes this special marking.

Note that the ability to re-activate a procedure is necessary in a self-stabilizing setting. For example, in our setting, only after the stabilization of the underlying unique leader procedure, we are sure that the actual procedure P is simulated uniquely (and thus correctly).

Given a code for a self-stabilizing bidirectional protocol P we now outline its unidirectional self-stabilizing simulation P':

```
Processor keeps: initial-state and current-state
Simulation Protocol:
if just-elected-leader then
    start the round simulation
        marking "restart" on special-token.
    just-elected-leader:= false.
    current-state:= initial-state.
if "restart marked token" arrives then
    if leader then remove "restart mark" off token
    otherwise: current-state:= initial-state.
repeat :
    execute Protocol for a round of communication.
    (starting from current state).
    update local state and output registers
        based on messages as in P.
    ended − round:= true.
```

Note that by the above procedures we are assured that eventually we have a leader that is elected the last time. It will start the start-firing-squad, that will restart. From then on it manages message transmissions that simulate rounds in P (which can be either a synchronous protocol or an asynchronous one). Now, since P is self-stabilizing and recovers from errors, performs restarts, and continuously checks the legal state maintenance– we are sure that once the message transmission phases are performed correctly, P' will stabilize as well, which completes the proof of Theorem 3.4. The performance claimed in the theorem is a result of the performance of the components and simply the linearity of expectation.

Acknowledgments

We are grateful to Umesh Vazirani for his very helpful remarks and suggestions concerning this paper.

References

[1] Y. Afek and G.M.Brown, Self-stabilizing over unreliable communication media. *Distributed Computing,* 7:1993, pp. 27-34.

[2] Y. Afek and E. Gafni, Distributed Algorithms for Unidirectional Networks, *SIAM J. on Computing*, 1994 pp. 1152-1178.

[3] Y. Afek, S. Kutten, and M. Yung, Memory-efficient self stabilizing protocols for general networks. In *Proc. of the 4th Int. Workshop on Distributed Algorithms*, Springer-Verlag LNCS, September 1990.

[4] B. Albert and A.P. Jayasumana, FDDI and FDDI-II: Architecture, Protocols, and Performance, *Artech House Telecommunication Library*, ed. V. Cerf, Artech House, 1994.

[5] E. Anagnostou, R. El-Yaniv, and V. Hadzilacos, Memory adaptive self-stabilizing protocols. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: Springer-Verlag LNCS*, November 1992.

[6] D. Angluin, Local and global properties in networks of processors, *Proc. 12th ACM Symp. on Theory of Computing* (1980), 82–93.

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

[8] B. Awerbuch and R. Ostrovsky, Memory efficient and self stabilizing network reset. In *Proc. of*

the 14th ACM Symp. on Principles of Distributed Computing, August 1994.

[9] B. Awerbuch, B. Patt-Shamir, and G. Varghese, Self-stabilization by local checking and correction. In *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, pages 268–277, October 1991.

[10] B. Awerbuch, and G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols, *Proc. 33nd IEEE Symp. on Foundations of Computer Science* (1991), 258–267.

[11] R. Cole, and U Vishkin, Deterministic Coin Tossing and Accelerating Cascades, *Proc. 18 ACM Symp. on Theory of Computing*, (1986), 206–219.

[12] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17:643–644, November 1974.

[13] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. of the Ninth ACM Symp. on Principles of Distributed Computing*, pages 103–118, August 1990.

[14] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. In P.G. Spirakis S. Toueg and L. Kirousis, editors, *Lecture Notes in Computer Science 579: Proceedings of the fifth Int. Workshop on Distributed Algorithms*, pages 163–180. Springer Verlag, October 1991.

[15] D. Dolev, M. Klawe, and M. Rodeh, An $O(n \log n)$ Unidirectional Algorithm for Extrema Finding In a Cycle. *J. of Algorithms*, 3: 245–260, 1982.

[16] S. Even, A. Litman, and P. Winkler. Computing with Snakes in Directed Network of Automata *Proc. 31st IEEE Symp. on Foundations of Computer Science* (1990), 740—745.

[17] G.N. Frederickson and N. Lynch, The impact of synchronous communication on the problem of electing a leader. *J. of the ACM* **34**(1) (1987), 98–115.

[18] T. Herman, Probabilistic self-stabilization, *Information Processing Letters* **35** (1990), 63–67.

[19] N. Honda and Y. Nishitani, The Firing Squad Synchronization Problem for Graphs, *Theoretical Computer Science,* v. 14, 1981, pp. 39–61.

[20] A. Israeli and M. Jalfon, Token management schemes and random walks yield self stabilizing mutual exclusion, *Proc. 9th ACM Symp. on Principles of Distributed. Computing* (1990), 119–130.

[21] A. Itai and M. Rodeh, Symmetry breaking in distributive networks, *Information and Computation* **88** (1990) 60–87.

[22] A. Itai, On the power needed to elect a leader *Proc. 4th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science*, Vol 486, Springer-Verlag, New York, (1989), 29–40.

[23] G. Itkis, and L Levin. Fast and Lean Self-Stabilizing Asynchronous Protocols. In *Proc. of the 35th IEEE Ann. Symp. on Foundation of Computer Science*, October 1994.

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

[25] S. Kutten, Stepwise Construction of Efficient Distributed Traversing Algorithm for General Strongly Connected Directed Networks, *The 9-th ICCC*, 446-458, 1988.

[26] N. Linial, Distributive Graph Algorithms–Global Solutions from Local Data. In *Proc. of the 28th IEEE Ann. Symp. on Foundation of Computer Science*, October 1987, 331–335.

[27] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung, Self-stabilizing symmetry breaking in constant space. In *Proc. 24th ACM Symp. on Theory of Computing*, May 1992.

[28] R. Ostrovsky and D. Wilkerson Faster Computation On Directed Networks of Automata. In *Proc. of Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (PODC-95), Ottawa, Ontario, Canada, August 20-23, 1995.

[29] R. Ostrovsky and M Yung, How to withstand mobile virus attacks, Proc. of the 10th ACM Symposium on the Principles in Distributed Computing, 1991, pp. 51-61.

[30] G. Parlati and M Yung, Non-exploratory self-stabilization for constant-space symmetry-breaking Proc. of the 2d ESA (European Symp. on Algorithms). Springer-Verlag, LNCS 855, 1994, pp. 183-201.

[31] G. Peterson, An $\Theta(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem. *ACM Tran. Prog. Lang. Syst.*, 4: 758–762, 1982.

[32] F.E. Rose, FDDI- A Tutorial, *IEEE Communication Magazine*, v. 24, pp. 10-15, May 1986.

[33] A.S. Tanenbaum, *Computer Networks*, 2-d ed. Prentice Hall Software Series, Prentice Hall.

[34] P. Tetali and P. Winkler, *On a Random Walk Problem arising in Self-Stabilizing Token Management* In *Proc. of the 10th ACM Symp. on Principles of Distributed Computing*, August 1991.

[35] G. Varghese. *Self-Stabilizing by Counter-Flushing* In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, August 1994.