Memory-Efficient and Self-Stabilizing Network RESET

(Extended Abstract)

Baruch Awerbuch* Rafail Ostrovsky†

August 15, 1994

Abstract

In this paper we consider the question of fault-tolerant distributed network protocols with extremely small memory requirements per processor. In particular, we show that even in the case of worst-case transient faults (i.e., in a self-stabilizing setting), many fundamental network protocols can be achieved using only $O(\log^* n)$ bits of memory per incident network edge. In the heart of our construction is a self-stabilizing asynchronous network RESET protocol with the same small memory requirements.

*Johns Hopkins University, Baltimore, MD 21218, and MIT Lab. for Computer Science. E-mail: baruch@blaze.cs.jhu.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARPA/Army contract DABT63-93-C-0038, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799.

†U.C. Berkeley and ICSI. Supported by NSF postdoctoral fellowship and ICSI. E-mail: rafail@cs.berkeley.edu.

1 Introduction

We examine the question of designing fault-tolerant protocols using only very small memory per processor. The type of fault-tolerance we require is so-called "self-stabilization", introduced by Dijkstra, which means, informally, that a protocol must be able to "recover" from an arbitrary transient fault. The type of memory constraints we impose is that in a network of n processors, each processor is allowed to have only $O(\log^* n)$ bits of memory per incident network edge. In this setting, we exhibit a variety of self-stabilizing protocols, including, for example, self-stabilizing spanning-tree, and self-stabilizing leader election. Our algorithms are asynchronous; they work for arbitrary network topology; they do not require unique processor ID's; they are uniform (i.e., every processor executes the same code); and they stabilize in polynomial time.

1.1 Self-stabilizing protocols

We consider distributed networks where processors and edges from time to time can crash and recover (i.e., dynamic networks), where additionally, when processors recover, their memory can be recovered in an arbitrary inconsistent state (to model arbitrary memory corruption). Despite this faults, we wish the network to be able to maintain and/or to be able to re-build certain information[1] about itself, whenever there is sufficiently long period of time without any faults.

The theoretical formulation of this model was put forth by Dijkstra [Dij74], who, roughly, defined the network to be "self-stabilizing" if starting from an *arbitrary* initial state (i.e., after any sequence of faults), the network after some bounded period of time (denoted as *stabilization time*) exhibits behavior as if it was started from a good initial state (i.e., stabilizes to a "good" behavior). Notice that this formulation does not allow any faults during computation, but allows an arbitrary initial state. Thus, if new faults occur during computation, it is modeled in a self-stabilizing formulation as if this is a new initial state from which the network again must recover.

It should be pointed out that the above model makes a distinction between the *code* stored in processor's hardware or non-volatile memory and which *can not* be altered and processor's *program state* (including things like program counter, variable values, etc.) which *could be corrupted* in an arbitrary manner. Thus, assuming that the code itself can not be corrupted, self-stabilizing algorithms recover from any transient fault, including not only crashes of processors and communication links but also arbitrary memory corruption. Notice that self-stabilizing protocols (by definition) do not require manual intervention in case of an arbitrary fault and do not require proper initialization to begin with (since they automatically recover.) In summary, self-stabilization is a very strong fault-tolerance property which covers many different types of faults and provides a uniform approach to the design of a variety of fault-tolerant protocols.

[1] For example, maintaining a spanning tree.

Due to this features, self-stabilizing protocols were used in the design many of existing systems. For example, self-stabilization was required for many DECNET protocols [Per83] including INTERNET [MRR80, Per83, Per85], and many IBM networks and protocols, including SNA [BGJ+85], PARIS/PLANET [CG88, ACG+90] and MetaRing [CO89, OY90]. Moreover, various aspects of self-stabilization were studied in the theoretical setting as well, including [BGW87, AG90, DIM90, KP90, AKY90, DIM91, APV91, Var-92, AKM+93].

1.2 Small memory and efficiency

An important consideration in the design of distributed protocols is in terms of efficiency, both in terms of execution/stabilization time and in terms of memory requirements per processor. Why execution/stabilization time is important is clear. Let us elaborate why memory requirements per processor are important. With the advent of fiber-optic media and high-speed networks, there is a technological trend to implement the protocols in hardware (preferably on a single *chip*). This allows mass-production of such devices, and fast processing of signals propagating through the network (i.e., propagating through high-speed hardware switches [CO89, OY90, MOOY92]). Moreover, the smaller the memory, the cheaper it is to manufacture. However, in order to achieve hardware solutions, one must get away from unique ID's (in order to make the production cheaper), make *uniform* protocols (i.e., the same hardware for all processors executing the same code) and reduce memory as much as possible. In this paper we propose uniform and efficient network protocols with extremely small memory requirements: $O(\log^* n)$ bits per incident network edge, where n is the number of processors.

1.3 Self-stabilizing RESET

A general methodology for making protocols self-stabilizing is to design a self-stabilizing RESET protocol which in case when something goes "wrong" can restart a system from a good initial state (see, for example, [AG90, KP90, APV91, Var-92].) Informally, the challenge (and the main technical difficulty) of designing a self-stabilizing RESET protocol is to ensure that even if something goes "wrong" *during the execution of the reset protocol itself*, this condition will be detected and corrected. Once this is accomplished, it is possible to design a spanning-tree protocol by designing a distributed self-stabilizing checker which verifies that spanning-tree is correct, and invokes a self-stabilizing RESET if it detects an error. Finally, self-stabilizing reset and self-stabilizing spanning-tree protocols can be used as a general "compiler" which turn network protocols which work only in static networks and only with proper initialization into ones robust against both memory and link failures [KP90, APV91, AV91, Var-92]. We show how all of this can be done using only very small memory per processor. Our main technical contribution is a memory-efficient and self-stabilizing RESET protocol:

MAIN THEOREM: There exists a deterministic self-stabilizing RESET protocol for arbitrary-topology asynchronous uniform network which requires $O(\log^* n)$ bits of memory per incident network edge and stabilizes in $O(n \log^2 n)$ time on networks with n processors.

Without fault-tolerance requirements, small-memory solutions for networks of identical nameless processors are much easier to construct. In particular, assuming that all processors are started in the same pre-specified initial state and assuming that there are no faults, we show simple randomized solutions which require only constant memory per incident network edge for leader election and spanning-tree algorithms. We then show how using our RESET protocol we can turn these algorithms into their self-stabilizing counterparts with only $O(log^* n)$ bits per incident network edge:

COROLLARY: There exists a randomized self-stabilizing spanning-tree and leader election algorithms for an arbitrary topology asynchronous uniform networks which require $O(\log^* n)$ bits of memory per incident network edge and stabilize in expected polynomial time, where n is the number of processors.

Once we have a self-stabilizing spanning-tree, the memory of the entire network can be used as a centralized memory arranged according to the in-order traversal of the spanning-tree, which allows us to convert centralized algorithms into distributed and self-stabilizing ones. We elaborate on this further after we preset our main construction.

1.4 Previous work and techniques

Seminal paper of Dijkstra [Dij74] introduced self-stabilization in a setting where each node can instantaneously read contents of memory of its neighbors and change its own configuration. The definition was extended to shared memory model in [DIM90] and to message-passing model in [AB89, KP90, APV91, Var-92].

[AB89] consider the question of establishing self-stabilizing reliable channel between a pair of processors over a physical channel. Further considerations to making a reliable channel between a pair of processors was given in [DIM91, APV91, Var-92]. In particular, [APV91] present a self-stabilizing Unit Capacity Data Link (UCDL), where only one packet can be in-transit in a Data Link. The reason, informally, why only one (or bounded) number of packets are allowed in transit over the channel is that this is a realistic assumption in practice, but also, assuming unbounded number of packets is problematic from theoretical viewpoint i.e., infinite number of packets in the channel can prevent stabilization indefinitely (for further discussion, see [DIM91].) In this paper, we assume a UCDL protocol (as a primitive) for every communication edge of the network.

A self-stabilizing RESET protocol was considered in [AG90, KP90, APV91, Var-92]. However, all the solutions presented there require at least logarithmic (and sometimes linear) amount of memory per processor. Other tasks, such as maintaining a spanning-tree, or leader election were also considered (see, for example, [KP90, AKY90, APV91, Var-92]). We stress, though, that they all require at least logarithmic space per processor. Subsequently,

[AIO92], suggested a recursive data-structure for reducing memory to *constant* size per edge. In order to implement the approach proposed there, one must first show how to iteratively unfold the recursion in [AIO92] in a self-stabilizing manner. It appears that the latter task has been accomplished by Itkis and Levin (private communication).

In the current paper, we provide the first self-stabilizing deterministic RESET protocol with only $O(\log^* n)$ memory requirements per incident network edge. Our protocol is *iterative*, it builds on a recursive data structure introduced in [MW86] and subsequently employed in [AIO92]. It also builds on the techniques developed in [APV91, MW86, AIO92, MOOY92, Var-92].

Another work which addresses small memory per processor requirements, is [MOOY92]. In particular, they assume (as a primitive) an ability to detect deadlock, and resolve (under this assumption) the question of token-management scheme on the ring using constant space per processor. They consider the model where processor can change state only when it receives a new message (see also [IJ90] for further discussion about this model.) In contrast, in our work we assume that every processor has a clock, but clocks of the processors are not synchronized. Thus, a processor can change state without receipt of a message from other processors. In this model, the question of deadlock detection using sub-logarithmic space per processor remained unresolved. As part of our construction, we show a solution to this problem as well (using the same small memory requirements.)

2 The model

2.1 Self-stabilization

In this paper we adopt the Input/Output Automata model (IOA), of [LT89], and closely follow the notation of [APV91, Var-92]. IOA is described by a *state set* S, *action set* A, and by its *step relation* $R \subseteq S \times A \times S$. An action a is *enabled* in state $s \in S$ if there exist $s' \in S$ such that $(s, a, s') \in R$. A subset of actions is denoted as "input actions" and another (disjoint) subset is denoted as "output actions". An *execution fragment* is an alternating sequence of states and actions $(s_0, a_1, s_1, \ldots)$, where $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. We say that an infinite execution fragment is *weakly fair* if in case when some action is continuously enabled, it is eventually taken. We consider only weakly fair executions. In a *timed execution* we have a time function t that associates with each action a_i time $t(a_i)$, such that t is non-decreasing and unbounded. The *duration* of state s_i is the time interval $[t(a_i), t(a_{i+1})]$. We say that some property holds in a time interval if it holds for each state in that interval. For stabilization time analysis, we normalize t so that each message is delivered within each time unit, and execution starts at time zero.

A *behavior* of an automaton is the set of sequences of external (input/output) actions generated by all the executions of A. A *problem* is a set of behaviors; an automaton is said to *solve* a problem Π if its set of behaviors is a subset of Π. There is a set of *initial states* specified for an automaton. Following [APV91], we say that

Definition 1 An IO automaton A is a *self-stabilizing* automaton solving problem Π, if for any fair behavior β of A (regardless of the initial state), there exists a behavior $\gamma \in \Pi$, such that there is a sequence α which is a suffix of both β and γ.

Informally, an automaton is self-stabilizing if after some finite time, it behaves correctly. The *stabilization time* of a behavior is t if after time t, the behavior is a suffix of a correct behavior. The stabilization time of an automaton is the worst-case stabilization time of all its fair behaviors.

The global state s of the network is a Cartesian product of the states of all the processors and the channels. (We remind the reader that both processors and channels are modeled by Input/Output Automata (similar to [APV91, Var-92]) and that the channels are in fact Unit Capacity Data Links (and processors know if the channel is operational or not). We define the set of *legal global states* as a set of states starting from which, the external behavior of the protocol is a suffix of the correct behavior. Thus, legal global states are the states where protocols have "stabilized".

In this paper, we deal with so-called "non-interactive protocols" [APV91, Var-92], where, the correctness can be specified by the I/O relation of the local inputs and the final topology of the graph (i.e. after faults stop). That is, with each processor V_i a local input I_i is associated. Given the final (quiescent) topology of the network $G = (V, E)$ and local inputs I_i for each node V_i of V, the *non-interactive task* is a task where the outputs O_i at each V_i is a function of $G = (V, E)$ and all the inputs I_i. (For formal definitions, see [Var-92].)

We deal with both deterministic and randomized protocols. (For leader election, for example, randomization is necessary, in order to break symmetry. However, our RESET protocol is deterministic.) When we consider randomized protocols, each processor has access to an independent source of randomness (modeled as a read-once tape, where if a processor needs to remember a bit it has to save it in its work memory.) In a randomized setting, we talk about expected stabilization time.

2.2 Distributed Checking

The goal of any self-stabilizing protocol is to detect if the system is in a legal global state, and if not, to "reset" the system into such a state. The two questions that immediately arise are how can a distributed system recognize if it is not in a legal global state, and if it recognizes that this is indeed not the case, how can it "reset" itself. We address both questions below.

How does a system verify if it is in a legal global state? The approach suggested in [KP90] is centralized: a leader takes a self-stabilizing *snapshot* (see [CL85]) of the entire network and then verifies (at the leader node) if the network is in a legal global state. If it is not, the leader starts a "reset" protocol (to be defined). Notice that this approach requires a linear (in the size of the graph) memory in the leader node. A different approach was suggested in [AKY90, APV91, Var-92], that of *local checking*. That is, every processor checks if its local "view" is "consistent" (i.e. satisfies certain set of predicates) for every incident edge. More specifically, [APV91, Var-92] define the notion of local checkability and

extensibility. Informally, this means that processors from time to time check their state and the state of their communication channels and neighbors for every incident edge. If the system is not in a legal global state then locally, no matter in what order processors check their edge/neighbors pairs, one of the processors will be able to detect while communicating with one of its neighbors on an edge e that one of the local predicates does not hold.

Informally, the notion of extensilbity is required to assure that even if we have a "moving" fault, it can not escape local checking. Once a fault is detected at any (or several) processors, a "reset" procedure is called. Formal definitions of local checkability and extensibility appear in [Var-92] and the reader is referred there.

For our purposes, local checkability and extesnability are not sufficient, and we generalize this notion to distributed checkability and extensibility (similar to the definition of [Var-92].) Briefly, the notion is as follows: every processor periodically checks if a given set of local predicates is satisfied for each edge/neighbor pair, as before. If any predicate is not satisfied, then as before it is guaranteed that the system is not in a good global state (hence a new "reset" must begin.) However, if it is satisfied, it does not guarantee that the system is in a good global state, rather, it guarantees that either:

(1) the system is in a good global state or

(2) after some bounded time, the system will be in a state where locally one of the local predicates on some edge is not satisfied.

Notice that if condition (2) holds, then we are guaranteed that *eventually* a "reset" will be called. Informally, this means that if something goes "wrong" then locally, one of the processors will eventually be able to detect it locally while communicating with one of its neighbors on an edge e.

3 Network reset

3.1 Problem statement

In this section we describe the reset problem, and exhibit a self-stabilizing RESET protocol. Let us review the definition of the *reset problem* defined in [AAG87, APV91]. We are given a network, with four commands that a RESET protocol supports at each node. In particular, at each node there is an input action to *receive* message m on edge e; and an output action to *send* message m along edge e. In addition, each node supports an output action *reset request* and an input action *reset signal*. The problem is to design a protocol ("reset service"), such that after the faults stop, if one of the nodes makes a reset request and no node makes infinitely many requests, then

1. In finite time all the nodes in the connected component of a requesting node receive a reset signal (*liveness*).
2. No node receives infinitely many reset signals (*termination*).

3. Let $e = (u, v)$ be any link in the final topology of the network. Then the sequence of $Send(m, e)$ input at u after the last reset signal at u is identical to the sequence of $Receive(m, v)$ output at v after the last reset signal at v (*consistency*).

Informally, (1) and (2) guarantee that every node gets a *last* reset signal, and (3) guarantees that the last reset signal provides a consistent reference time-point for the nodes.

3.2 Non-stabilizing reset

Recall that a RESET is a procedure which allows a node to "push a reset" button and cause the entire network to go to an initial state. In [AAG87] a *constant* space per edge solution is presented to this problem for *dynamic networks*, however the solution is *not* self-stabilizing. Our starting point is their algorithm, which we give a high-level description of below:

The node can be in three states: "normal-execution", "freeze" and "unfreeze". Each node has a list of edges which are currently operational and can check the state of neighboring nodes. If the node (locally) wishes to reset it goes from a "normal-execution" state to the "freeze" state and sends "freeze" messages to all its (non-frozen) neighbors. (That is, is sends "freeze" messages to all the neighbors, but the ones which are currently frozen ignore this message and send a "reject" message on this edge).

When a node which is not frozen receives a "freeze" message it marks the edge from which it has received a freeze message as a parent edge and sends a "freeze" message on all other edges. Thus, when "freeze" propagates it creates a tree (or a forest) rooted at nodes that invoked the algorithm. All the "frozen" messages wait for an acknowledgment message from all their children (those which did not send "reject" message) before sending an acknowledgment message to a parent. Thus, when a node receives the acknowledgments (ACKs) to all "freeze"" messages it sent, it is guaranteed that all the nodes it introduced into the algorithm have been frozen.

We hold the nodes of the tree frozen until the root of the tree has received all its pending ACKs. At this moment we know that no branch of the tree is propagating, and we may unfreeze the tree going from the root outward, converting nodes into "normal-execution" state. This may be accomplished by "unfreeze" messages from parents to sons in the tree. (In a dynamic network, if an edge to a "frozen" node recovers, a "freeze" message is sent to it. When some edge fails (in a dynamic setting) while the node is frozen, it is considered as if this edge was not there to start with. That is, if a "freeze" or an ACK were expected on it, they may be considered as if they had been received.) Moreover, if e led from the node to its parent in the tree, then the node henceforth considers itself as an originator of the "freeze" algorithm.) It was shown in [AAG87] that the above algorithm is a good "reset" service for the dynamic networks.

3.3 Self-stabilizing reset with logarithmic space

The above algorithm is not self-stabilizing since the "freeze" tree could have a cycle. That is, "Parent" edges can form a cycle. In the algorithm of [APV91] they show that by maintaining

a "distance" variable at each node, such that node's "distance" is one more than its parent's "distance", the above problem can be resolved. Specifically, "distance" is initialized to 0 upon reset request, and its accumulated value is appended to "freeze" messages. In [APV91] it is proved that with the above simple modification, the resulting protocol is a self-stabilizing version of RESET. Notice, however, that their solution requires to maintain distance from the root, which requires $O(\log n)$ memory per processor.

3.4 From Self-stabilizing Reset to Self-stabilizing Deadlock Detection

Suppose processors are either connected in a line or a cycle graph. In its simplest form, deadlock detection is an ability for processors to eventually correctly output whether they form a line or a cycle. In this section, we show a memory-preserving reduction from self-stabilizing reset to this very simple self-stabilizing deadlock detection, as we elaborate below.

The reason [APV91] solution requires logarithmic memory per processor is due to the "distance" variable which is set while reset is propagating (recall that the rest of the code requires only constant space per incident network edge.) We start by modifying the [APV91] protocol as follows: when a node receives a "freeze" message from its parent, it sends a "freeze" message to only one of its neighbors and waits for a response from this neighbor before sending to other neighbors.

With this modification, the solution still requires logarithmic amount of memory, since the "distance" variable requires *logarithmic* amount of space at each node. This is so, since initiated reset request always starts as 0, and to ensure that there is no cycles, the nodes maintain the numbers in the increasing order, where node numbered i can wait (for an ACK) only from a node numbered $i+1$ (and fails and recovers the link if this is not the case). Thus, "distance" numbering is thus organized as follows: $(x_1), \ldots, (x_{n-1}), (x_n)$ where each parenthesis represents a processor keeping a variable x_i, where $x_1 = 0$ (i.e., a root) and for each x_i, $(x_{i-1} + 1) = x_i = (x_{i+1} - 1)$ is a local constraint which every processor can (locally) verify. If any constraints are not satisfied a new reset is initiated. Notice that the above numbering can be constructed during propagating "freeze" message and ensures that there is no deadlock. Thus any scheme which can reduce memory requirements in the above scheme will directly yield a reduction in the memory requirements needed for our self-stabilizing reset protocol.

3.5 The Log-Star Acyclicity Checker

In the previous subsection, we use $\log n$ bits "distance" variable at each node to guarantee acyclicity. We now show how to do this using only $\log^* n$ bits of memory. (That is, recall from the previous subsection that our objective is to make sure, in a self-stabilizing manner, that there is no deadlock during propagating "freeze".) We first describe our log-star data structure (which extends [MW86, AIO92]) and then show that it is in fact self-stabilizing.

More specifically, we exhibit a self-stabilizing data-structure which is growing dynamically during propagating "freeze" massages.

At each processor, we keep $\log^* n$ *levels*. Each processor contributes the same constant number of bits for every level. Each level keeps certain "numbering" information which we describe below.

On the first level, the "root" (i.e. initiator of RESET) is initialized to 0. Moreover, in the first level memory slot of each processor, processors are numbered as an alternating 0/1 sequence:

$$(0)(1)(0)(1)\ldots$$

In order to check that they are not in an odd-length cycle, processors continuously copy the value of the right-hand neighbor and make sure that their value is equal to the value of the right-hand neighbor plus 1 (mod 2). If any processor detects that there are two consecutive 0's or 1's it starts a new copy of "reset". Finally, observe that this (first) level guarantees that if processors *are* in a cycle, then it is of even length (as in an odd-length cycle this is impossible, since otherwise there must be two consecutive numbers that are identical.)

At level 2, we have two-bit variables which are written distributively, where 0/1 bit of the first level is used as a *pointer* to indicate which (of the two bits) of the "distributed" 2-bit number each processor keeps at the second level. Notice that using two-bit binary numbers we can count up to four in binary:

$$\underbrace{(0)(1)}_{(0,0)}\underbrace{(0)(1)}_{(0,1)}\underbrace{(0)(1)}_{(1,0)}\underbrace{(0)(1)}_{(1,1)}\underbrace{(0)(1)}_{(0,0)}$$

Processors verify that the second level is divided into pairs according to the marking in the first level. In order to perform distributed counting up to four, processors in the second level perform string copy and binary addition (of two-bit numbers in binary) to verify that the second level distributed numbering is correct. If this is so, then again we know that either processors are arranged in a line or in a cycle which is divisible by 8.

In order to implement distributed checking, we define several tokens (each token requires a constant number of bits to implement). One token copies a distributed two-bit number to the next pair of processors (we specify how this is done below.) Another token performs binary addition (to make sure that the second level has consistent 2-bit distributed numbering (mod 4):

$\left(\overset{\text{copy}}{\underset{\text{token}}{\longrightarrow}}\right)$	$\left(\overset{\text{copy}}{\underset{\text{token}}{\longrightarrow}}\right)$	$\left(\overset{\text{copy token}}{\underset{\text{add-one token}\downarrow}{\longrightarrow}}\right)$	$\left(\overset{\text{copy token}}{\underset{\text{add-one token}\downarrow}{\longrightarrow}}\right)$
(0	0)	(0	1)

All the other $\log^* n$ levels are done similarly using different tokens for each level and for each pair of numbers (between parenthesis). Parenthesis (i.e. delimiters) are checked as sequences of all one's from the previous level. For example, in the third level, we use as a delimiter the ending of a $(1,1)$ sequence of the second level. In general, we use the sequence of all one's from the previous level as a delimiter of the next level. Notice that a special token can recognize the delimiter of the next message without keeping track of the number of the level. (This is done by a third token which scans distributed number and checks if all bits are 1.)

$$\underbrace{\underbrace{(0)(1)}_{(0,0)}\underbrace{(0)(1)}_{(0,1)}\underbrace{(0)(1)}_{(1,0)}\underbrace{(0)(1)}_{(1,1)}}_{(0,0,0,0,0,0,0,0)}\underbrace{(0)(1)}_{(0,0)}(0)$$

For each level, we must verify that the numbering is in an increasing order (with wrap-around). As mentioned above, while the first level numbering can be verified and established by processors using local checking, the second level and higher level numbering must be verified and established distributively.

The question arises how the distributed numbering is established in the first place, and how it is verified. As mentioned above, we do so by doing bit by bit string copy and binary addition. The string-copy is done by a token which goes back and forth and copies one bit at a time. (We remark that the token needs only constant number of bits to represent. It keeps track which bits are being currently copied by marking appropriate places in both strings. The token keeps track if it found a special bit-copy marker each time it goes left or right. If no marker is found and the token reached its boundary and must turn, then a new "reset" is initiated. Also, recall that the existence of a boundary is guaranteed by a previous level.) Add-one is done using binary adder with the help of another token. (This token is also represented using constant number of bits, including "carry" bit, for example.)

The following invariants (i.e. predicates which every processor periodically verifies for each edge) are maintained by each processor for each level and each token:

1. *orientation of the line*: direction towards the root (either left or right is maintained) and agrees with direction towards the root of left and right cell.
2. *copy token is not lost*: direction of the current location of the token for string copy.
3. *binary adder token is not lost*: direction of the current location of the add-one token.
4. *building or testing*: After a single string-copy, add-one and check-marker are completed, DONE variable is set to true.
5. *boundary token is not lost*: direction of the current location of the marker token (which checks delimiter for the next level).
6. *bit copy marker is not lost*: a bit-copy token keeps track if it found a bit-copy marker before it changes direction.

The first invariant ensures that the only possible configuration of a "waiting-for" ACKs graph is either a line or a cycle (i.e., all nodes agree on the orientation towards the root node.)

The second and third variable ensures that locally processors agree in which direction the relative position of every token is (i.e., to their left or to their right). Since we are guaranteed acyclicity by the smaller levels, this ensures that if a token is lost, either there will be local disagreement in which direction the token is located, or the node which "should" have the token will detect that it is missing. In both cases a new version of "reset" is initiated. The forth invariant is used to show that the numbering is consistent (i.e., once set, if numbers change, this will be detected.) The fifth condition guarantees that markers for the next level are set properly and do not change once set. Finally, the sixth condition guarantees that string-copy is functioning properly (a similar condition is also made for binary-adder token).

Each processor verifies the following conditions for every level, and if these conditions are not satisfied, "cuts" this edge (i.e., processor "fails" and then "recovers" this edges, thus causing a new "reset" to begin): (1) direction towards the root agrees with your neighboring processors; (2) direction of the token (for string-copy and add-one) for each step agrees with the direction to the left and right neighbors for each step and is changed only when token of this step goes over; (3) Once *done* is set to "true" the numbering bits of the local processor and the delimiters for the next level do not change. (4) if string-copy token encounters a root node (i.e. reset originator node) it verifies that corresponding number (for the given level) is a string of zeros. In addition, processors verify that boundaries of the next level are established exactly when a sequence of 1's appears in the previous level and that string-copy and binary addition tokens find consistent markings after "done" is true.

Finally, notice that if the first level works properly (i.e., all of the above constraints are satisfied) and there is no "reset" it guarantees that either processors are in a line or in a cycle which is divisible by 2. The second level guarantees that it is a line or a cycle of length divisible by 8. More generally, if levels 1 through i do not detect an inconsistency, then either processors are in a line or in a cycle which is divisible by $f(i)$, where $f(1) = 2$; $f(i)_{i\geq 2} = f(i-1) \cdot 2^{f(i-1)}$. Thus, if all $j = O(log^* n)$ levels work properly, then we are guaranteed that there is no cycle.

We now specify how this $\log^* n$-level enumeration is incremented (to include one more node) during propagating "freeze" message. The basic idea is to ensure that higher levels of the data-structure work only after lower-level ones have stabilized. Thus, we first complete a new segment of the first level, and once it is completed, we "fill-in" the distributed bits of the second level and so forth. Below, we show several steps of how this numbering is propagating during "freeze":

step 1: (0)

step 2: $\underbrace{(0)(1)}_{(0,0)}$

step 3: $\underbrace{(0)(1)}_{(0,0)}(0)$

...

step 7: $\underbrace{(0)(1)}_{(0,0)}\underbrace{(0)(1)}_{(0,1)}\underbrace{(0)(1)}_{(1,0)}(0)$

step 8: $\underbrace{\underbrace{(0)(1)}_{(0,0)}\underbrace{(0)(1)}_{(0,1)}\underbrace{(0)(1)}_{(1,0)}\underbrace{(0)(1)}_{(1,1)}}_{(0,0,0,0,0,0,0,0)}$

Notice, for example, that the second level does not appear all at once, but rather every 2 steps, with the initial "frontier" with only first level filed in, etc. Moreover, we ensure that the first level has finished a local testing (invariant four) before we fill-in the second level. The third level is filled in when we encounter $(1,1)$ of the second level, and so forth.

Finally, we note that the running time of the largest level dominates (where the largest level is the longest). However, the size of the largest level in order to count up to n must be $\log n$ and to do a sting copy (and binary addition) takes quadratic time (in $\log n$). Thus, the stabilization takes $O(n \log^2 n)$ time (with a more elaborate scheme we can do somewhat faster stabilization, we postpone this to the final version.) The rest of the analysis is similar to [APV91].

4 Rooted Spanning-Tree and Other Extensions

Assuming *proper initialization and no faults*, it is easy to design a randomized leader election protocol, using only *constant* space per incident network edge. Moreover, we can simultaneously elect a leader, by building a *rooted* spanning tree, where the leader is a root: To begin with, every processor is a leader. Then, a standard elimination *tournament* protocol is run, where remaining leaders play asynchronous tournaments and subsume each other (leaders flip coins and if one leader flips a 1 and the other flips a 0, then a "zero" leader and its "zero" tree joins the tree of a "one" leader. This assures that the last remaining leader is never killed. The communication is done in an asynchronous manner via trees of each leader in a standard fashion, where trees do handshaking.)

To make this protocol self-stabilizing, we note that our acyclicity checker needs $O(log^* n)$ bits per incident edge and can be constructed during tree-growth and ensures that every intermediate (and final) tree is indeed a tree and does not have cycles. If cycles are detected, we run our RESET protocol to start a new execution.

Having a self-stabilizing rooted spanning-tree algorithm allows us to organize memory of the entire graph as a Turing-machine tape embedded in the in-order traversal of the spanning-tree, as was advocated in [OY90, AIO92]. This centralized machine can then compute any non-interactive task (see section 2.1) as long as the entire memory of the network is sufficient for the centralized solution of this task. (This is so, since this centralized machine can compute for any two nodes in the network if they are connected by the operational edge, thus implementing "adjacency matrix oracle" with the same small memory requirements. Without self-stabilization, and assuming proper initialization and no faults, this oracle can be implemented in a strait forward manner using only constant space per edge, however, in order to make it self-stabilizing, we need $O(\log^* n)$ bits of memory per edge in order to implement a self-stabilizing spanning-tree.) This, in turn, allows us to compile any randomized sequential

algorithm which requires $O(M)$ memory into a distributed and self-stabilizing version of it, as long as every processor has at least $O(\log^* n)$ memory per incident network edge, and the total joint memory all the processors is also $O(M)$. That is, assuming that every processor has at least $O(log^* n)$ memory per edge, we get a general compiler which allows us to convert an arbitrary centralized probabilistic graph algorithm which gets as its input topology of a network into its fault-tolerant, memory-preserving and distributed implementation by the network itself.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.

[AB89] Y. Afek and G. Brown. Self-stabilization of the alternating bit protocol. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[ACG+90] B. Awerbuch, I. Cidon, I. Gopal, M. Kaplan, and S. Kutten. Distributed control for Paris. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 145–160, 1990.

[AG90] A. Arora and M. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AKM+93] A. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese Time Optimal Self-stabilizing Synchronization STOC-93.

[APV91] B. Awerbuch, B. Patt, and G. Varghese, Self-stabilization by local checking and correction. FOCS-91.

[AV91] B. Awerbuch, and G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols, FOCS-91.

[AIO92] B. Awerbuch, G. Itkis and R. Ostrovsky Hardware-Based Self-Stabilization, unpublished manuscript, April 29, 1992.

[AKY90] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, Italy, 1990.

[BGJ+85] A. Baratz, J. Gray, P. Green Jr., J. Jaffe, and D. Pozefski. SNA networks of small systems. *IEEE Journal on Selected Areas in Communications*, SAC-3(3):416–426, May 1985.

[BGW87] G. Brown, M. Gouda and C.Wu A self-stabilizing token system. *In 20'th Hawaii International Conference on System Sciences*, 1987.

[CG88] I. Cidon and I. S. Gopal. Paris: An approach to integrated high-speed private networks. *International Journal of Digital & Analog Cabled Systems*, 1(2):77–86, April-June 1988.

[CO89] I. Cidon and Y. Ofek. Metaring - a full-duplex ring with fairness and spatial reuse. Research Report RC 14961, IBM, Sept. 1989. Also INFOCOM 90.

[CL85] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.

[Dij74] E. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. PODC-90.

[DIM91] S. Dolev, A. Israeli, and S. Moran. Resource bounds on self-stabilizing message driven protocols. PODC-91.

[Fin79] S. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.

[KP90] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. PODC-90.

[IJ90] A. Israeli, M. Jalfon Token Management Schemes and Random Walks Yield Self Stabilizing Mutual Exclusion PODC-90.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3), September 1989.

[MW86] S. Moran and M.K. Warmuth "Gap Theorems for Distributed Computation" PODC-86.

[MOOY92] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung Self-Stabilizing Symmetry Breaking in Constant-Space STOC-92.

[MRR80] J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Commun.*, 28(5):711–719, May 1980.

[OY90] Y. Ofek and M. Yung. Principles for high speed network control: lossess-ness and deadlock-freeness, self-routing and a single buffer per link. PODC-90.

[Per83] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7:395–405.

[Per85] R. Perlman. An algorithm for distributed computation of a spanning in an extended LAN. In *Proceedings of the the 9'th Data Communication Symposium*, pages 44–53, September 1985.

[Wec80] S. Wecker. DNA: the digital network architecture. *IEEE Transactions on Communication*, COM-28:510–526, April 1980.

[Var-92] G. Varghese, Self-Stabilization by Local Checking and Correction, MIT LCS Ph.D. Thesis, 1992.