Fast and Lean Self-Stabilizing Asynchronous Protocols*

Gene Itkis

Technion, Israel and Boston Univ.
Itkis@cs.technion.ac.il

Leonid Levin

Boston Univ.† and Hebrew Univ.
Lnd@bu.edu

Abstract

We consider asynchronous general topology dynamic networks of identical nameless nodes with worst-case transient faults. Starting from any faulty configuration, our protocols self-stabilize any computation in time polynomial in the (unknown) network diameter. This version sacrifices some diversity of tasks and efficiency for simplicity and clarity of details. Appendix gives more efficient procedures in less detail.

1 Introduction

Networks can resist asynchrony by each node keeping a step counter restricted to $0, \pm 1$ difference over edges (i.e. advancing when no neighbor is behind). It is often reduced mod 3 (we call it *slope*) if no self-stabilization required. Faulty configurations, however, can have inconsistent mod 3 counter: some cycles unbalanced, with more up edges than down. Slope has much greater utility when *centered*, i.e., has a unique node, *leader*, with no down edges. It then yields a BFS tree the construction/maintenance of which is known to self-stabilize many basic network management protocols (we generalize this experience to all randomized linear space problems). Initiating an (uncentered) slope is an easy task and Sec. 3 gives a simple fast deterministic algorithm for it (BFS with a few precautions). For simplicity, it takes larger (log of diameter) space per node than $O(1)$ as done in Appendix.

The main task, leader election (i.e. modifying any slope into a centered one), is much harder and known to be impossible for deterministic algorithms. Our main result gives a fast randomized algorithm for it, using one byte per node and a pointer to a neighbor.

The third protocol, *Interface* (using no additional space) runs the first two as subroutines. It assures that any (consistent with it) variation in either of the first two protocols cannot affect the other one.

Smart distributed networks perform many organizational tasks with various costs and assumptions. One can represent the network topology by a connected graph G given (say, as an adjacency matrix) on a read-only input tape. Then the computational power of any network with total memory S is in the obvious class $Space(S)$. Our protocols assure that this trivial upper bound *can* be reached, not only by the optimistic models but also by much more realistic ones.

These models are *asynchronous* (i.e. no global clock exists, all processors have separate, uncoordinated clocks) and *dynamic* (i.e. the network can change in the runtime of the protocols). Moreover, the protocols are *self-stabilizing*, (this strong kind of transient fault tolerance is discussed below).

The use of randomness is essential, since no deterministic protocols can elect a leader starting from a configuration with a symmetry between existing leaders [Dij74].[1] Deadlocks (absence of leaders) detection can be deterministic: no need to break symmetry.

1.1 Self-stabilizing protocols

A *self-stabilizing* protocol works "correctly" no matter what state it is initiated in. This implies highly desirable fault tolerance: resilience against *worst-case* transient errors and dynamic changes. Much theoretical and practical work has accumulated since the pioneering work [Dij74] by Dijkstra. Self-stabilization (at least partial) is an important component in the existing networks, and has been a central issue in the

*FOCS-94. Supported by NSF grant CCR-9015276, Guggenheim Foundation and Israel Council For Higher Education.

†Comp. Sci. Dept., 111 Cummington St., Boston, MA 02215.

[1] [GL82] shows that even in a most general model of computation, no deterministic algorithm can reduce the input symmetry by a non-constant prime factor.

distributed computation research and other areas (see for example [AKY90, APV91, AV91, DIM90, DIM91, KP90, Var92, A$^+$93, M$^+$92, GKL78, K78, G86] and the bibliographies therein).

Two general approaches were developed. [KP90] proposed to self-stabilize any protocol by periodically collecting distributed snapshots of the system in some central node, which then locally decides if the configuration captured by the snapshot is inconsistent, and if it is, resets the protocol. A major drawback of this technique was the centralization[2]; collecting the complete configuration required that at least one node be as large as the network.

Another general technique introduced by [AKY90] in their Spanning Tree protocol replaced the global by local checking. [AV91], [Var92] used it to develop compilers converting synchronous deterministic protocols into self-stabilizing versions. These compilers are less general than those of [KP90], e.g. cannot automatically handle randomized protocols. But [AV91] used the technique on some problems (e.g. Leader Election). The key idea was to combine local checking with (synchronized) re-executing of the protocol.

[M$^+$92] advocates the practical importance of constant space per node protocols. But the methods relying on local checking cannot be extended in a natural way to use sublogarithmic space per processor — e.g. with constant space the local neighborhoods of an inconsistent global configuration can look the same as local neighborhoods of a consistent one. In fact, [IJ90] proved a logarithmic lower bound on space for self-stabilizing deadlock detection even on a ring. This led to a commonly held belief that the general self-stabilization with sublogarithmic space per processor is impossible. However, this lower bound applies only to a more restrictive model. In it a processor can make transitions only if it has a "token", determined as a predicate of the neighborhood. Then, even a ring of n processors has a deadlock configuration unless each processor has $\Omega(\log n)$ memory. Thus, no self-stabilizing token management (preventing deadlocks) is possible. If the "no deadlock" property is guaranteed externally, [M$^+$92] gives a randomized constant space protocol for token management on a ring in a message passing model. The lower bound of [IJ90] no longer applies if all processors act all the time (and computing the token predicate is no easier than computing the transition function).

A recent result of Awerbuch, Itkis and Ostrovsky [I$^+$92] gives randomized self-stabilizing protocols using $\log\log n$ space per edge for leader election, spanning tree, network reset and other tasks. It was improved to constant space per node for all linear space tasks by Itkis, and Itkis, Levin [I$^+$92, IL92][3]. The present paper is a detailization of [IL92]. These constructions were later modified in [AO94] to extend the scope of tasks solvable deterministically in $O(\log^* n)$ space per edge (beyond forest/slope construction, for which our algorithms were already deterministic).

[2] As can be seen from the structure of our solution, the mere existence of the central node significantly simplifies the task.

2 Model, Interface, theorems

2.1 Model

We consider a distributed network of diameter d. Each node v communicates (via edges) with its *neighbors* $w \in \mathbf{E}(v)$ in the system's connected undirected reflexive *communication graph* $G=(V,E)$. Each node's state consists of bits and pointers to immediate neighbors. The bits of a node x are visible to any its neighbor y as well as whether a pointer of x points to x or to y. Nodes can detect a neighbor with a given property of state, set a pointer to it,[4] and change state based on all above information.

Asynchrony is modeled by *Adversary* determining a sequence of nodes with arbitrary infinite repetitions for each. The nodes act in this order. A *step* is a time interval until each node acts again at least once.

To define self-stabilization let each processor have the following fields: read-only *input*, write-only *output*, and read/write *work* and *structure*. A *configuration* at time t is a quintuple $\langle G, I, O_t, W_t, S_t\rangle$, where functions I, O_t, W_t, S_t on V represent the input, output, work and structure fields respectively. The standard protocol running in S_t and the computation running in the other fields are independent and interact only via reading the slope fields of S_t. A problem P is defined as a set of the correct i/o configurations $\{\langle G,I,O\rangle\}$. A deterministic protocol solves P with *self-stabilization* in t_s steps if starting from any initial configuration, for any time $t > t_s$ the configuration $\langle G, I, O_t\rangle$ satisfies P. One cannot tolerate the worst case transient faults having actual halting configurations: the system could *start* in a wrong one.

Deterministic protocols cannot reach our goals and must flip coins. Each node v has a sequence of "coin" bits $\mathbf{coin}(v)$ as read-once input. It may be foreseen or even skewed by Adversary. We only assume that

[3] using hierarchical constructions (Lemma 4 below) similar to those developed originally by [K78] and [G86] in the context of cellular automata.

[4] Proposition 2.1 requires deterministic choice of neighbor, e.g. the first qualified, if the edges are ordered.

starting from any step i no nodes get over $O(\log |Vi|)$ identical bits. But any larger bound only stretches the Main Theorem time proportionally.

Our protocols are Las Vegas. Since they do not halt, this means that after *stabilization* output is independent of the subsequent coin-flips. Stabilization is the repetition of the non-S_t configuration after the slope stops changing. The Las Vegas *stabilization period* is then the expected stabilization time (from the worst case configuration). Slightly more general definitions considered in the literature will be accommodated in the final versions.

2.2 RSpace and centered slope

Proposition 2.1 *Let P be a problem on a network G with a centered slope. A self-stabilizing randomized asynchronous protocol solving P on G with $O(s)$ space per node exists if and only if $P \in RSpace(s|V|)$. The protocol (stabilizes and) runs in time $d^{O(1)}$ times the (known) upper bound of the RSpace algorithm.*

We consider sequential TM but the arguments are easy to extend to parallel computations. E.g., equally fast we can simulate one step of a tape of cellular automata and even one sweep of a TM head throughout the tape without changing direction.

A centered slope yields an obvious spanning tree structure, through the up edges. So, a TM tape can be embedded on a DFS tour of the tree.[5] A read-only (input) tape containing the adjacency matrix[6] of G can be simulated as follows. To read the entry (v, w) of the adjacency matrix, find node v and mark it. Then find node w and see if there is an edge coming from a marked node. In the end, clear the mark of the node v. A single look-up of the adjacency matrix can thus be simulated in $O(|V| \log |V|)$ time. This time can be further improved to $d^{O(1)}$.

So far we showed how to simulate a step of TM from its arbitrary configuration. We may add a pointer at each cell pointing towards the head, to guarantee its existence and uniqueness. However, the network may be initialized in some configuration corresponding to a legal but unreachable configuration of TM. Detecting this condition is in general impractical. Instead, we augment the TM with a clock containing a sufficient amount of bits. The number of bits must be at least logarithmic in the running time of the *RSpace* algorithm and given, as an input or a simple function of the tree size. Whenever the clock overflows, the TM work fields are re-initialized and its computation is restarted[7] (so if the computation is randomized it is important that the output does not depend on the coins — only the running time does).

2.3 Interface

Here we present *Interface* running two subroutines: *Leader Election* (LE) and *Slope Initiation* (SI) (Fig. 1-3). The access by LE and SI to a node v is regulated by field $v.\mathsf{ctl} \in \{open, closed\}$ (see interface rules). SI initiates a slope in fields $v.\mathsf{h3} \in Z_3^+$. Then LE elects a unique leader: makes the slope centered.

Let $v.\mathsf{h3}$, $(v.\mathsf{h3}–w.\mathsf{h3} \bmod 3) \in \{-1,0,1\}$. Then *variance* of a path $v_0 \dots v_k$ is the sum $\sum_{i=0}^{k-1}(v_{i+1}.\mathsf{h3} - v_i.\mathsf{h3} \bmod 3)$. So, a slope is an assignment of $\mathsf{h3}$ fields with all cycles balanced, i.e. of zero variance.[8] It defines a consistent partial orientation: a neighbor $w \in \mathbf{E}(v)$ is *under* v (and v is *over* w) if $v.\mathsf{h3} \equiv w.\mathsf{h3}+1 \pmod 3$. The edge vw points *down* and wv *up*. This test $\mathbf{up}(wv)$ is the only function of slope used by LE.

Interface prevents LE from breaking slope correctness. However, SI still must keep and update the evidence of it. To ease the updating, LE marks the set of *roots* (potential leaders). LE may create temporary local minima with no path down the slope (to a root). So, it supplies additional data to guide the more efficient versions of SI to a root. These are *float* flag, denoted $v = F$, subordinate to the slope, and $\mathbf{rank}(v) \in \{0,1,2,3,4,5\}$, superordinate to the slope.

Predicate $\mathbf{root}(v)$ tests if v is a *root*, defined as $\mathbf{rank}(v) = 0$. A closed root is called a *crash*. Procedure $\mathbf{crash}(v)$ and predicate $\mathbf{cr}(v)$ (for "closed root") make and test crash at v.

Interface rules: SI, LE can $\mathbf{crash}(v)$. LE makes no other interface changes on a closed v, or without closing it, or under roots, or near crashes. LE may change **rank** or, unless in root or over a node, increment $.\mathsf{h3}$. SI may open v except a root over non-roots, and decrement $.\mathsf{h3}$ of a crash not under non-roots.

LE is designed to run jointly with SI through the interface. But, if slope correctness is guaranteed then a "fake" SI suffice which just crashes non-roots under roots and opens nodes when permitted by Interface.

[5] The (patented) idea to embed a tape (ring) in a spanning tree from [OY90] was pointed out to us by R. Ostrovsky.

[6] The output must be correct for any numbering the graph nodes used in the matrix.

[7] Necessity of such re-computing is argued in [AV91].

[8] A weaker condition suffices for most applications: the absence of long (especially, cycling) chains of up edges (contributing a delay factor). The max length of such chains can change by at most $2d$ factor in any time period without crashes.

2.4 Main theorem

We will prove the following statements about each algorithm running jointly with an adversary which acts as permitted by Interface to the other algorithm.

SI *stabilization* period is the longest time unbalanced cycles or crashes exist without adversary (acting as LE) making new crashes. SI *response* period is the longest time a node remains closed in absence of crash nodes. After SI has stabilized it has no effect whatsoever (except for the response period delays).

$v=F$ is interpreted as "slightly lowering" $v.\mathsf{h}3$, so $\mathbf{Up}(vw) \stackrel{def}{=} \mathbf{up}(vw) \vee (v.\mathsf{h}3 = w.\mathsf{h}3 \;\&\; w=F\neq v)$. Ranks indicate direction (down on odd, up on even) towards a root along *forward* edges vw: $\mathbf{fwd}(vw) \stackrel{def}{=} [\mathbf{rank}(v) < \mathbf{rank}(w)] \vee [\mathbf{up}(vw) \,\&\, \mathbf{rank}(v) = \mathbf{rank}(w) \in \{2,4\}] \vee [\mathbf{Up}(wv) \,\&\, \mathbf{rank}(v) = \mathbf{rank}(w) \in \{1,3,5\}] \vee [v = w \,\&\, \mathbf{root}(v)]$. Any forward cycle is a (possibly reversed) **Up** cycle.

Lemma 1 (Crash) *After 1 step LE creates no new crashes and any node v has a forward edge.*

Lemma 2 SI *responds in* 1, *stabilizes in* $d+4$ *steps.*

Theorem 1 (Main) *In* $d^{O(1)} \log |V|$ SI *response periods after slope stabilizes, LE makes it centered.*

2.5 Roots, pointers, LE theorems

LE maintains a pointer to a neighbor (self, iff root). Sec. 4.1 will define *legal* edges and pointers. In particular, any legal pointer is forward. The legality cannot be broken by actions permitted to SI. Root–root edges are legal. LE crashes roots over non-roots and any node with illegal edges/pointer. LE never creates such nor crashes other nodes.

Proof of Lemma 1 (Crash). Neither adversary, acting as SI, nor LE can break legality. So, LE crashes make each edge/pointer legal and cease within one step. LE cannot create non-crash roots. LE pointer edges are forward. ■

Assume now SI has stabilized. Then, a path of up edges is always the shortest: otherwise closing it with a shorter path forms an unbalanced cycle. Forward edges cannot increase rank, so their cycles are unbalanced. By Lemma 1, forward edge paths can terminate at roots only. So, after SI stabilization any forward path leads to a root. Since no new roots can now be created (roots are created as crashes only) there is a root r_0 which stays root forever. Define *height* $h(v)$ of node v as the variance of paths from r_0 to v plus d (to assure $0 \le h(v) \le 2d$). A node v is called *grounded in* root r if there is an up edge path from r to v. When (and only when) $v.\mathsf{h}3$ is incremented (by LE), v enters F. *Floating* refers to entering/exiting F. F do not appear in roots or under non-F. Say, a root *belongs* to non-F nodes grounded in it.

A node v is called *overrooted* if some of its neighbors lack some of its roots. A node could only loose a root by some node (including itself when entering F) floating on its up path from the root, or by root dying (changing rank and pointer). Only the later is permitted by Interface. Similarly, the only way v can acquire a root r is by exiting F, i.e. only if there are no F under v. But then there must be a grounded non-F neighbor under v which thus has been overrooted before v exited F. So, "overrootedness" only propagates up (increasing height). Roots of (non-F) neighbors are called *linked*. A node is called *idle* when it is a root not flipping (popping) **coin**(), or is neither grounded nor floating, or is an F with no F under it.

Main Theorem Proof. After 1 step forward paths lead to roots, which thus exist. The poly-d bound on idle time is computed in Sec. 5. Corr. 4.4 proves the identity of coin flips of linked roots. Assuming the adversary cannot so distort the coin flips as to make two roots flip $\gg \log |V|$ identical coins from some step on, the following Claim yields Main Theorem 1.

Claim 2.2 *Let no node be idle for t steps and no linked roots flip k coins. Then at most one root remains after* $2td(k+4)$ *steps.*

Proof. Any path between roots has overrooted nodes. Let v be lowest (min h) overrooted. No grounded F have height $\le h(v)$: otherwise, overrooted nodes exist closer to the root. So, any set of roots acquired by v's neighbors includes the roots of v. While v stays overrooted it keeps at least one root, say r. Within kt steps, all neighbors of v are ungrounded (and float over v in $2t$ steps) or grounded in r. All (also ungrounded) F under them float in t more steps. Within another t steps they exit F (acquiring the roots of v). Thus, in $t(k+4)$ steps v seizes to be overrooted. The least height of overrooted nodes may increase $< 2d$ times, which implies the Claim. ■

3 Slope Initiation lemma

We assume now LE creates no roots. Slope Initiation checks (and creates, if broken) the slope and some certificate of its correctness. The simplest certificate would be a map of net nodes into the sequence of consecutive integers ("actual heights"). The map must preserve edges and labels mod3. Running alone, SI

Command	Test $[w \in \mathbf{E}(v)]$	Action at v (if Interface allows)
1. Zeroing	$\mathbf{low}(v) \vee \mathbf{root}(v)\,\&\,\forall w\,[\mathbf{root}(w) \Rightarrow w_*\mathsf{h} \geq v_*\mathsf{h} > 0]$	$\mathbf{crash}(v)$; $v.\mathsf{h} \leftarrow 0$
2. BFS	$\exists w : w_*\mathsf{h} < v_*\mathsf{h} - 1$	$\mathbf{crash}(v)$; $v.\mathsf{h} \leftarrow \min_w w_*\mathsf{h}+1$
3. Match h_3, h		$v.\mathsf{h}_3 \leftarrow (v_*\mathsf{h} \bmod 3)$; $v.\mathsf{h} \leftarrow v_*\mathsf{h}$
4. Exit	$\forall w\ (w.\mathsf{h}_3 \equiv w_*\mathsf{h} \bmod 3\ \&\ \lvert w_*\mathsf{h} - v_*\mathsf{h}\rvert \leq 1)$	$v.\mathsf{ctl} \leftarrow \mathit{open}$

Figure 1: *Slope Initiation.* $\mathbf{low}(v) \stackrel{def}{=} \neg\mathbf{root}(v)\,\&\exists w : [(w_*\mathsf{h} > v_*\mathsf{h}\,\&\,\mathbf{root}(w)) \vee (v_*\mathsf{h} = w_*\mathsf{h}\&\mathbf{up}(wv))]$.

could repair such map (if broken) with simple BFS. Also our *SI* must conform to Interface and take a few precautions: to tolerate any adversarial actions of *LE*, permitted by Interface.[9]

This simple protocol for *SI* is given in Fig. 1. It has a field $v.\mathsf{h}$ interpreted as height (possibly outdated). Function $v_*\mathsf{h}$ (updated height) equals $v.\mathsf{h}$ for crashes and exceeds $v.\mathsf{h}$ by $(v.\mathsf{h}_3 – v.\mathsf{h} \bmod 3) \in \{0,1,2\}$ for other nodes. Then $v_*\mathsf{h} \equiv v.\mathsf{h}_3 \pmod 3$ for any non-crash v. (Interface may delay $v.\mathsf{h}_3$ updates for BFS'ed crashes.) Call v a *zero* if $v_*\mathsf{h} = 0$, and an edge vw *long* if $|v_*\mathsf{h} - w_*\mathsf{h}| > 1$. Say, v is h-*over* $w \in \mathbf{E}$ if $v_*\mathsf{h} > w_*\mathsf{h}$.

A local minimum root must be a zero: otherwise it is (crashed and) zeroed (tr.1m). So are non-roots h-under roots (tr.1c) or over but not h-over crash (tr.1h). BFS crashes and shortens long edges (tr.2). Whenever the interface allows, $v.\mathsf{h}_3, v.\mathsf{h}$ are matched (tr.3, possibly by two $.\mathsf{h}_3$ decrements). Similarly, a node is opened when there are no $v.\mathsf{h}_3, v.\mathsf{h}$ mismatch or long edges in its neighborhood (tr.4).

SI uses $O(1)+\log d$ space since nodes on distance i from a zero[10] can gain at most $2i$ height (by induction on i) in any time interval. If a node is initialized in a higher space s it stays within space $s+O(1)$ eventually dropping it to $O(1)+\log d$.

Claim 3.1 *SI creates no zeros (tr.1) after 2 steps.*

Proof. The initial long edges with a non-crash lower end are reversed or made short (by BFS) within the first step. *SI* can only decrease $v_*\mathsf{h}$. It is increased only when *LE* increments $v.\mathsf{h}_3$ which has no neighbors under. Thus, after the first step, long edges can be created only by *SI*, all with a crash lower end.

[9]The integer sequence requires log space per node. Appendix A.1 explains how a special bit sequence α can be used instead. Like integer line, α cannot have loops and allows fast (log time) detection of errors.

[10]Each step the slowest node makes one action. It can be used instead of zero until permanent zero appears in the first two *SI* steps: After any *LE* step, forward paths end in roots or unbalanced cycles, thus, containing long edges adjacent to roots. The lowest roots are zeros.

LE creates no new roots, and *SI* creates them now only h-over other roots (BFS) or as zeros (tr.1). So, all local $_*\mathsf{h}$ minima roots are zeros after the next step.

Tr.1c,h can only be triggered within first two steps by edges created in the first step: for tr.1c by BFS to the non-root under long edge; for tr.1h by BFS or tr.1 on a node h-over a long edge to a non-root. ■

Claim 3.2 *BFS (tr.2) terminates within $d+2$ steps.*

Proof. If no crashes or long edges exist, the Lemma is proven. Otherwise, after the second step there is a zero. Let $k > 0$ be the smallest such that after $k+2$ steps there is a long edge vw, $w_*\mathsf{h} > v_*\mathsf{h} < k$ (by Claim 3.1, v is a root). After the second step, only BFS (tr.2) creates long edges. So, after $k+1$ steps there was a long edge uv, $v_*\mathsf{h} > u_*\mathsf{h} < k-1$, contradicting the minimality of k.

Let vw be a long edge closest to a zero after $d+2$ steps. Then its higher end w has a path to a zero which is a long edge wv followed by $< d$ short edges, so $v_*\mathsf{h} < d$ which contradicts the above. ■

Without long edges, all cycles are balanced. After $d+2$ steps no new crashes appear.

Claim 3.3 *After $d+4$ steps all roots are open.*

Proof. Consider a crash v with $v.\mathsf{h}_3 \not\equiv v_*\mathsf{h} \bmod 3$ after $d+2$ steps. Suppose there is a non-crash w over v. But then $w_*\mathsf{h} \neq v_*\mathsf{h} + 1$. So, either $w_*\mathsf{h} \leq v_*\mathsf{h}$ (cannot happen after the first step) or $w_*\mathsf{h} > v_*\mathsf{h} + 1$ (making vw long: cannot occur after $d+2$ steps). So, no such w may exist, and tr.3 is not blocked by the interface. After no long edges or $.\mathsf{h}_3, .\mathsf{h}$ mismatches or roots h-over (and thus over) non-roots exist, any crash is opened within one step by tr.4. ■

So, after $d+4$ steps *SI* only opens (immediately) any closed node without any other changes and the h_3 labeling is a consistent slope, which proves Lemma 2.

4 Leader Election

Besides slope (and $.\mathsf{ctl}$), at each node v, *LE* uses only two fields: $v.\mathsf{p}$ and $v.\mathsf{s}$. $v.\mathsf{p}$ points at one of the neigh-

$v.\mathsf{s}$	C_0,D_0,B_0 F_0,F_1,E_0	E_1	C_1 E_2	D_1 E_3	B_1 A
$\mathbf{rank}(v)$	5	4	3	2	1

v	$\neg F_0$	D	C,E	C_0,E_{12}	E_0	root
$\mid$	$\mid$	$\mid$	$\mid$	$\mid$	$\mid$	$\mid$
w	F	B	B,D,A	C_1	C	any

Figure 2: *LE* ranks and safety. Pointer $v.\mathsf{p}=w$ must have either $\mathbf{root}(v)$ & $v\in\{A,B,C,D_0\}$ or $\mathbf{fwd}(vw)$. Also, for non-root $v\in\{A,B,C,D_0,E_2,F_1\}$, $\mathbf{up}(wv)$. Unsafe edges: $(B,D_0),(B_0,D)$; and up from w to v, as above right.

$\mathbf{infect}(w,v) \stackrel{def}{=} v\in\mathbf{E}(w)$ & $v\in\{C,D_0\}$ & $[w=A \vee (w=C_1 \& v=C_0) \vee (w\in\{E,D_1\} \& \mathbf{up}(vw))]$

Order	Test (required for some $w\in\mathbf{E}(v)$, preferably, $w=v.\mathsf{p}$)	Action (if safe v results)
1. Float	$\mathbf{up}(wv)$ & $[w\neq E \vee v=E_0]$	$v.\mathsf{p}\leftarrow w$
		$v(\{C,D\}\mapsto E, *\mapsto E_0 \stackrel{\mathsf{h3}++}{\mapsto} F\mapsto F_1)$
2. Basic	$w\in\{A,C_1\} \vee [\mathbf{root}(v) \& (v=D \vee \mathbf{coin}(v))]$	$v(\{D,F\}\mapsto A, B_0\mapsto C_1);$ $v.\mathsf{p}\leftarrow w$
cycle	$\nexists u$: $[\mathbf{infect}(v,u) \vee \mathbf{infect}(u,v) \vee v=A \& u=F_1 \& \mathbf{up}(vu)]$	$v(A\mapsto B_1\mapsto B_0\mapsto C_0, C\mapsto D_0)$
3. Infect	$\mathbf{infect}(w,v)$	$v(D\mapsto D_1, C\mapsto E);$ $v.\mathsf{p}\leftarrow w$

Figure 3: *Leader Election.* $v(X\mapsto Y, X' \stackrel{\mathsf{h3}++}{\mapsto} Y',\ldots)$ changes v: X into Y, X' into Y' (also incrementing $v.\mathsf{h3}$), etc.; the entire action is un-applicable to other states. $\mathbf{coin}(v)$ returns the next random bit.

bors of v and $v.\mathsf{s}$ is in one of 13 states listed and ranked in Fig. 2. We drop states' indices where irrelevant and $.\mathsf{s}$ in formulas like $v.\mathsf{s}=A$. Assume *SI* stabilized. Node v is *above* w (w is *below* v) if there is a path of up edges from w to v. No node is above itself.

Say, $v_k\ldots v_0 (v_i.\mathsf{p}=v_{i-1})$ is a p-*chain*; and node v *leads to* w if there is a p-chain from v to w.

4.1 Safe and legal edges

LE crashes illegal edges according to sec. 2.5. It also has an internal *safety* version of legality (Fig. 2), and a state A used akin to crash. An edge/pointer is *legal* if it is safe, or root–root, or up edge from A, or producible from those by crash and/or lowering the non-lower root end. A node is legal/safe if all its edges and pointer are. *LE* changes safe nodes to safe and, if legal, unsafe nodes to A, preferably non-root.

No non-roots are under roots now. Fig. 2 implies these properties of safety: (i) Unsafe A–A edges are from roots down; (ii) Entering legal A preserves safety of edges, except upward or to unsafe roots; Unsafe legal node can legally change to A, if it is (iii) a root or (iv) all its root neighbors are A or safe; (v) Setting pointer down from A to A preserves safety; (vi) Legal pointers are safe unless to unsafe non-A roots.

Claim 4.1 *All edges are safe after* $2d+2$ *steps.*

Proof. Nodes change only to safe or (legal) A. Safe roots have no down edges and cannot become unsafe (ii). Unsafe roots become A within a step (iii), making all pointers safe (vi). After next step any root to non-root edge is safe: roots remain A until safe; non-roots are safe or enter A, saving the edge (i). So, after 2 steps only two kinds of unsafe edges are possible: r-edges (A-root to A-root up) and n-edges (A-non-root to non-A-non-root up). The interface (and (v)) allow saving the highest r-edges, by turning the upper pointer down, say to the lower end of the r-edge. So, the max height of the r-edges decreases. Unsafe n-edges become safe (A–A) in a step (iv). Any n-edge has n-edges under it the previous step: otherwise, it becomes safe and must (ii) remain so. Thus min height of n-edges increases each step. ■

From now on assume there are no crashed or unsafe nodes or unbalanced cycles. *LE* cannot create those. Any p-chain leads to a root.

4.2 States: intuition, mnemonics

The transitions of *LE* are given in Fig. 3. Altering $v.\mathsf{ctl}$, as required by Interface, is left implicit.

The *basic* cycle starts with a root spreading up an "activating" signal A, which changes into a "strong back" state B_1 when there is nowhere else to propagate. The backing is completed by entering "weak back" B_0 (from leaves down to the root). After collecting all B_0 signals, the root chooses at random a "coin" C_0 or C_1 and spreads it up. This is acknowledged by "done" D_0 and the cycle is repeated. This

cycle is similar to Dijkstra's 4-state token ring [Dij74].

Any two roots will, at some point, broadcast different signals. When this is detected, *infection* starts: the "weaker" nodes enter (from C) "exit" states E, or (from D_0) "infected done" D_1 (a basic cycle state, as it may return to A without floating). Infection propagates down and kills the root(s) below. Then *floating* starts, gently elevating ungrounded nodes, until grounded. F is entered (from E_0) if and only if $.\mathsf{h}3$ is incremented. F may return to A when there are no F under it (signaled by entering F_1).

A root is *dying* if there is an E above it. A node is *dying* if all roots below it are. Dying nodes enter F (float), before re-entering A.

4.3 Linked roots

Call v, w *in synch* while their sequences of $C_{0,1}$ states (except possibly the first and last) are identical.

Claim 4.2 *A node is in synch with its roots. Dying roots never again flip coins.*

Proof. Dying roots float before flipping any coins. Indeed, F have no roots, so let $v \neq F$. Only A,B are safe over B_0 and only A,D over D_0. Any transit from $\{A,B\}$ to $\{A,D\}$ either holds A or includes $B_0 \mapsto C \mapsto D_0$. If v holds A, **infect** blocks $C \mapsto D_0$ under it. So, while r does $B_0 \mapsto C \mapsto D_0$ any non-float v over (and by induction above) it does the same. E must float (having no down edges) before entering C.

Only C,E are safe under C. While v exits and enters C, it must pass through A. So, any non-dying node under (and by induction below) v does the same or dies (if in C, as v passes through A).

If $v = C$ and w is below v, then (from vertical compatibility) $w.\mathsf{s} = v.\mathsf{s}$ or w is dying within a step. ■

Claim 4.3 *Basic states neighbors are in synch.*

Proof. Call an edge *live* if its both ends are in basic states, except edges $(A,C),(C_0,C_1)$ (dying in one step, tr.3). The following is the exhaustive list of transitions between live edge states: $(C,C) \mapsto (C,D) \mapsto (D,D) \mapsto (D,A) \mapsto \{(A,A),(B_1,D_1)\} \mapsto (A,B) \mapsto (B,B) \mapsto (B,C) \mapsto (C,C)$. So, obviously the sequence of the C_0, C_1 states in the two ends are identical except possibly first and/or last. ■

Corollary 4.4 *Linked roots have identical coin flips except possibly the first and last two.*

5 Idle times

Here we prove a poly-d bound on any node's idle time (not optimal, but simpler to prove). By $v(S \xrightarrow{t} T)$ we denote that if v is in a state of S (any, if $S = *$) then it is or within t steps will enter a state of T.

5.1 Floating

Claim 5.1 *If v is a rank 5 non-root, then* $\mathbf{up}((v.\mathsf{p})v)$ *and $v = \mathrm{E}_0 \vee v.\mathsf{p} \neq \mathrm{E}$ within $t_f \stackrel{def}{=} 8d$ steps.*

Proof. The claim is satisfied within a step (by tr.1) if $v \in \{B_0, C_0, D_0\}$ or v has a down edge vw such that either $v = E_0$ or $w \neq E$. Otherwise, v must enter/exit E_0 to satisfy the claim. $v.\mathsf{p}$, $(v.\mathsf{p}).\mathsf{h}3$ cannot change without satisfying the claim. v may exit or (since F cannot point up) enter E_0 at most twice while $v.\mathsf{p}$ keeps $.\mathsf{h}3$.

Entering/exiting E_0 can be blocked by nodes under (but then the claim is satisfied in a step). Otherwise, exiting can be blocked only by down pointers from non-E, and entering by pointers from E_0. Consider the tree of all p-chains of alternating $(\neq E)/E_0$ leading to v. If the tree contains only v then in a step v can enter/exit E_0 or claim be satisfied.

The tree depth is $\leq 2d$. No nodes can join the tree (nor change branches) until v enters/exits E_0: pointers with both ends of max (5) rank are set only by tr.1; and such pointer edges $u\vec{w}v$ can change states to have exactly one end in E_0 only by w entering/exiting E_0.

Also, leaves leave the tree in a step: a leaf can either change pointer or enter/exit E_0. So in $2d-1$ steps only v remains in the tree and in $2d$ steps either claim is satisfied for v or v enters/exits E_0. ■

Claim 5.2 *If $v \in \{\mathrm{D},\mathrm{E}\}$ for $\geq 3d$ steps, then only max rank and* D,E *nodes lead to v and only tr.1 may change pointers on nodes leading to v.*

Proof. Call a node w *sterile* if $w \in \{D_1, E_{123}\}$, no C,D_0 exist under w and no C_1 point at it.

Only tr.1 can set pointers on a sterile w: tr.2 sets pointer only on A,C_1; tr.3 requires either $w \in \{A,C\}$ or C,D_0 to be under w (both contradicting sterility).

$\{C,D_0\}$ come only from B_0, unsafe under D,E. C_1 can point only down, is unsafe over D and cannot change pointer to E. So, a node can loose sterility only by changing to A or E_0 (D_1, E_{123} can change to no other states); And any $w \in \{D_1, E_{123}\}$ is sterile within a step: by tr.1 C_1 pointing at $w(=E)$ either changes pointer or enters E, and by tr.3 C,D_0 under w are infected and change to D_1,E also within a step.

Max rank children of sterile w can decrease rank only if changing pointer. Non-max (1–4) rank children

u of a sterile node w can only be in D_1,E_{123} ($u = C_1$ contradicts sterility of w; the others' rank is either 5 or too low), and so are sterile within a step.

Let $v \in \{D_1,E_{123}\}$. Consider the tree of all non-max rank p-chains leading to v (i.e. containing only $w \in \{C_1,D_1,E_{123}\}$ and of depth at most $3d$: d for each rank). Sterile tree nodes can loose sterility only by exiting the tree: A cannot lead to D,E, and E_0 has max rank. And non-max rank children of sterile nodes become sterile in a step, and no new such children appear. Thus, the minimal depth of non-sterile tree nodes increases each step. ■

Corollary 5.3 *Let* $t_d \stackrel{def}{=} 8d+(3d+2)t_f$.
Then $v(\{\mathrm{D_1,E,F}\} \xrightarrow{t_d} \{\mathrm{A,F}\})$ *with* $\mathbf{up}((v.\mathsf{p})v)$.

Proof. E_0 is safe under any node and over only E. $w(* \mapsto E)$ (tr.1) preserves safety of $w.\mathsf{p}$ and of up edges. It may be prevented by safety of down edges to non-E; $w(F \mapsto E_0)$ is also prevented by pointers from E_0. Nothing else prevents the change to E. E can only change to F with no nodes over.

Let $v \in \{D_1,E\}$ and let $w \notin \{D_1,E\}$ be a highest such below v. By safety, $w \notin \{B,F\}$. If $w = A$ then $v(* \xrightarrow{d} A)$ (E is unsafe over A, and $D \mapsto A$ by tr.2). Finally, $w(\{C,D_0\} \mapsto \{E,D_1\})$ by tr. 3. So, either after d steps only D_1,E (thus, no roots) remain below v or in $2d$ steps v enters A.

Let only D_1,E be below v. After $3d$ steps only D,E or max rank nodes may lead to (and only tr.1 may change pointers on) each of them (Claim 5.2). Existing pointers from max rank are down in t_f steps (Claim 5.1). New ones can be set only down (tr.1; Claim 5.2). Pointer from non-max rank cannot be set on E,D_1 by tr.1. So, if $w \in \{D_1,E_{123}\}$ is the last such leading to or below v, $w(\{D,E\} \xrightarrow{t_f+1} E_0)$ (tr.1). Thus, in $7d+3dt_f$ steps v and all below it enter E_0. Pointers on E_0 must be down and those from $\neq E_0$ disappear in t_f steps (Claim 5.1; pointers on E_0 can be set only down from E_0, tr.1). Now, maximal down paths from v get shorter each step (tr.1). So, in $8d+3dt_f+t_f$ steps v enters F.

If $v = F$ then $\mathbf{up}((v.\mathsf{p})v)$ within t_f steps (Claim 5.1). Since $\mathsf{h3}$ changes only when entering F, v enters F as $\mathbf{up}((v.\mathsf{p})v)$ becomes true. ■

Corollary 5.4 *Ungrounded nodes float in* t_d+d *steps.*

Proof. If $u \neq F$ is ungrounded, then each lowest non-E below u changes to E (tr.1), so within d steps u (and all below it) are E. $E \xrightarrow{t_d} F$ by cor. 5.3.

Let $u = F$ be ungrounded. Consider $u' = F$ below (or equal) u with $w.\mathsf{p} = u'$, $w = E_0$ (by safety, $w.\mathsf{h3} = u'.\mathsf{h3}$). Such pointer can result only from u entering F and becoming above u'. Otherwise, the transition must be at u' or w. But tr.2,3 cannot result in a pointer with such endpoints; tr.1 sets pointers only down; and w could not enter E_0 (no other state can safely point at F from the same $\mathsf{h3}$); if u' enters F it increments $\mathsf{h3}$ floating from below u. So, while u stays F, no new such edges $\vec{wu'}$ can be created for any u' (= or) below $u = F$. The existing such edges disappear in t_f states (claim 5.1, E_0 is unsafe over F). After that, each lowest non-E below u enters E in a step (tr.1, similar to above). So, ungrounded F float in $t_f + d$ steps. ■

5.2 Basic cycle

Claim 5.5 $v(\{\mathrm{A,B}\} \xrightarrow{t_a} \{\mathrm{C,E_0}\})$, *for* $t_a \stackrel{def}{=} t_d+5d$.

Proof. $v(A \xrightarrow{2} B_1)$. Indeed, let $v = A$ and for $w \in \mathbf{E}(v)$ let (i) $\mathbf{up}(vw)$ and $w \in \{D,F_1\}$, or (ii) $w \in \{C,D_0\}$ (and $\neg\mathbf{up}(vw)$). If no such w, $v(A \xrightarrow{1} B)$. If (i) then $w(\{D,F_1\} \xrightarrow{1} A)$ (tr.2). If (ii), $w(\{C,D_0\} \xrightarrow{1} \{E,D_1,A\})$). New such w do not appear: due to the order in Fig. 3, new F_1,C,D as above change to A,E,D_1 within the same transaction.

$v(\{A,B\} \xrightarrow{t_d+3d} B_0)$. Indeed, let $v \in \{A,B_1\}$. The pointers of A,B_1 never change; A is entered only with the pointer down on A, unless in root (tr.2). So, in each 2 steps the lowest A leading to v enter B. Their min height cannot decrease: new such A can appear only over the existing ones. So, after $2d$ steps no A lead to v, unless v exited and entered A, passing through C or E_0. In t_d steps more all E,F pointing at and D_1 adjacent to B_1 leading to v change to F with pointer down (or to A not leading to v; Cor. 5.3). C cannot safely point at B. So then maximal p-chains of B_1 leading to v get shorter each step (tr.2; $B_1 \mapsto B_0$ preserves edge safety and pointer safety from the B, pointers on the last B leading to v are now only down from max rank nodes, and so remain safe too).

Finally, let $v \in \{A,B\}$. At any time, let set x consist of v and all A,B,D nodes below v. D_0 is safe only under $\{A,D,F\}$ and immediately infected under A,D_1. So, D_0 cannot be entered under A,B,D_1 and, in a step, highest D_0 below v enter D_1 (tr.3), or A (tr.2 if there is an A under it) or E (tr.1). In d steps x contains no D_0. Then, no new nodes enter x ($\{A,B,D_1\}$ are entered only from F, unsafe below v, or from D_0). In t_d+3d steps more, all nodes in x enter B_0 (or leave x). Now each lowest node of x leaves x in a step (tr.2, C is safe wherever B_0 is, except over elements of x). So, within d more steps $v \in \{C,E_0\}$. ■

Claim 5.6 *$v(\{\text{C,D}\} \xrightarrow{t_c} \{\text{A,E}\})$ for $t_c \stackrel{def}{=} 2dt_a$.*

Proof. $r(C \xrightarrow{2dt_a - d} \{A,E\})$, for root r. Indeed, let root $r = C$. It can change only to D (and then immediately to A) or to E. If any node above r enters E then it, and thus r, will float in t_d steps (cor. 5.3). Let set x at any moment contain r and nodes w above r, $w.\mathsf{s} = r.\mathsf{s}$ (only C,E are safe under/below C,E; so any down path from $w \in x$ to r contains only C, or r floats in t_d steps). The min height of B over x increases each t_a steps (Claim 5.5; new B, coming from A, cannot appear over C; new elements of x come only from B over x). So, after dt_a steps no such B remain, and no new nodes join x (assuming r has not changed). After t_a steps more no B are adjacent to x (claim 5.5, new B come from A which would infect the adjacent C of x), and no pointers on x can appear, except from max rank down (new C come from B and old weaker C_0 are already infected, so only tr.1 can set pointers on x, only down and only from x or max rank). In t_d steps more, E pointing on x enter F with pointer down (cor. 5.3). Now only C over can prevent nodes from exiting x, so in d more steps x contains only $r \neq C$.

Let $v \in \{C,D_0\}$. In $2dt_a - d$ steps either v is ungrounded (and all pointers on v or below are down, cor. 5.3) or some root of v is A. B,F are unsafe under C,D,E. So, the highest $w \notin \{C,D,E\}$ below v (if any) must be A (under D). Thus, the max height of A or min height of C,D below v increases each step. ■

Corollary 5.7 *No node is idle for $t_a+t_c+2t_d+d$ steps.*

Proof. v is idle when it is a root not flipping coins ($\leq t_a+t_c$ steps: claims 5.5, 5.6), or an F with no F under it, or ungrounded but not floating. Let $v = F$ with no F under it. No F can appear under v, so v exits F_0 in 1 step. F_1 enters A whenever A appears under it (A cannot exit under F_1). Any node under v will be E or A in t_a+t_c steps (claims 5.5, 5.6). E float from under v within t_d steps and ungrounded v floats in t_d+d steps (Cor. 5.4). ■

Acknowledgements.

The presentation of our result benefited from the comments made in 1992 by Manuel Blum and Moti Yung, and in 1994 by Oded Goldreich and Yishay Mansour. We express our warm gratitude to them.

The first author thanks B. Awerbuch and R. Ostrovsky for the collaboration on the first sublog-space LE algorithm [I+92] and for the opportunity to explain to them in 1992 a preliminary version of the results of this paper. Also a lecture by B. Awerbuch provoked the interest of the first author in the area.

References

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. *Workshop on Distributed Algorithms*, 1990. Springer-Verlag (LNCS 486).

[AK93] S.Aggarwal, S.Kutten, Time optimal self-stabilizing spanning tree algorithms. *FSTTCS*, 1993.

[A+93] B.Awerbuch, S.Kutten, Y.Mansour, B.Patt-Shamir, G.Varghese. Time optimal self-stabilizing synchronization. *STOC*, 1993.

[AO94] B.Awerbuch, R.Ostrovsky. Memory-efficient and self-stabilizing network reset. *PODC*, 1994.

[APV91] B.Awerbuch, B.Patt-Shamir, and G.Varghese. Self-stabilization by local checking and correction. *FOCS*, 1991.

[AV91] B.Awerbuch, G.Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. *FOCS*, 1991.

[Dij74] E.W.Dijkstra. Self stabilizing systems in spite of distributed control. *CACM*, 17, 1974.

[DIM90] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *PODC*, 1990.

[DIM91] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election. *5th Workshop on Distributed Algorithms*, 1991.

[G86] Peter Gács. Reliable computation with cellular automata. *J. of Comp. System Sci.*, **32**, 1, 1986.

[GKL78] P. Gács, G.L. Kurdiumov, L. Levin. One-Dimensional Homogeneous Media Dissolving Finite Islands. *Probl. Inf. Transm.*, 14/3, 1978.

[GL82] Peter Gács, Leonid Levin. Causal Nets or What Is a Deterministic Computation? *Information and Control*, **51**, 1982.

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. *PODC*, 1990.

[I+92] Gene Itkis. Self-stabilizing distributed computation with constant space per edge. Presented at colloquia at MIT, IBM, Bellcore, CMU, ICSI Berkeley, Stanford, SRI, UC Davis. 1992. Includes joint results with B. Awerbuch and R. Ostrovsky, and with L. Levin.

[IL92] Gene Itkis, Leonid Levin. Self-stabilization with constant space. Manuscript, Nov. 1992 (submitted to *STOC*93; Also reported by L. Levin in ICALP Tutorial Lecture, July 1994. Later version: Fast and lean self-stabilizing asynchronous protocols. TR#829, Technion, Israel, July 1994.

[K78] G.L. Kurdyumov, An Example of a Nonergodic Homogeneous One-Dimensional Random Medium with Positive Transition Probabilities. *Sov. Math. Dokl.*, **19**, 1, 1978.

[KP90] S. Katz, K. Perry. Self-stabilizing extensions for message-passing systems. *PODC*, 1990.

[M+92] A. Mayer, Y. Ofek, R. Ostrovsky, M. Yung. Self-stabilizing symmetry breaking in constant-space. *STOC*, 1992.

[OY90] Y.Ofek, M.Yung. Principles for high speed network control: losses-ness and deadlock-freeness, self-routing and a single buffer per link. *PODC*, 1990.

[Ro71] R. Robinson, Undecidability and non-periodicity for tiling a plane, *Invencione Mathematicae* 12: 177-209, 1971.

[Thu12] A.Thue. Uber die gegenseitige Lage gleicher Teile gewisser Zeichenreichem. *Kra.Vidensk.Selsk.I. Mat.-Nat.Kl.*, 10, 1912. Also in: A.Thue. *Selected Math. Papers*. ed.: T.Nagell, *et al.* Universitetsforlaget, 1977.

[Var92] G.Varghese. Self-stabilization by local checking and correction. PhD thesis, MIT, 1992.

Appendix

A $O(1)$ space Slope Initiation

The algorithms below use in each node $O(1)$ bits and pointers to neighbors. A simpler implementation (detailed in sec. A.4) runs in time $|G|^{O(1)}$. A more careful (sketched in sec. A.5) implementation stabilizes (and responds) in time polynomial in d and degree.

The Slope Initiation of sec. 3 used a map of net nodes into the sequence of consecutive integers, which requires $\log d$ space per node. Below we describe a special bit sequence α which can be used instead. Like integer line, α cannot have loops and allows fast (log time) detection of errors.[11]

Namely, α is an $O(1)$ bits[12] per node *no-loop* certificate, such that any string of the form ss has an illegal subsegment of length $O(\log |s|)^2$. A *loop-detector* algorithm will run permanently in constant size special fields. It will not affect other fields when (its fields and) the certificate is consistent. It will discover and break in poly-log parallel time pointer cycles. Some algorithms will use modified α preserving the loop-detecting property. Also polylog segments of α encode information similar to the height in log space SI.

First, sec. A.1–A.3 consider Slope Initiation on graphs of degree bounded by 2 (i.e. cycles and simple paths). In the case of a simple path, any .h3 assignment is a valid slope, and the task is trivial.

Lemma 1 guarantees forward edges. We extend definition of forward to include edges to roots from crashes and upper roots. SI at each node v uses forward pointer $v_*\mathsf{p}$ (somewhat similar to $v.\mathsf{p}$ of LE). Say v is a *zero* if $v = v_*\mathsf{p}$ (corresponding to zeros of sec. 3).

Loop detector will catch and break a zero-less cycle Then to guarantee the cycle balance, a stronger property will be assured: any path from zero has variance ≥ 0. Each zero-zero path in the cycle graph simulates a single TM (Prop. 2.1) checking/restoring its balance and absence of nodes under zeros.

A.1 No-loop certificate: claims

Now we discuss detecting/breaking of zero-less ($_*\mathsf{p}$-) cycles. This will be central to the SI implementation.

Pointer chain acyclicity is simple to certify by a sequence of increasing integers. However, we look for constant space certificates. No such certificates

[11] Handling α requires awkward bit programming, so some intuitive but tedious details will fit only in the journal version.

[12] Actually, 2 bits per node suffice, but we use more to avoid non-essential coding issues.

can be 1-step verifiable [IJ90]. (Interestingly, a two-dimensional analogue — aperiodic tiling — is possible [Ro71].) But we present $O(1)$ space certificates verifiable on poly-logarithmic neighborhoods.

A slower-working certificate could be the Thue (or Thue-Morse) sequence $\mu(k) \stackrel{def}{=} \sum_i k_i \bmod 2$, where k_i is the i-th bit of k [Thu12]. It has no overlapping segments, which are equal except for a shift (thus no substrings of the form sss). Thus, any loop can be detected after three rounds. Our sequence $\alpha(k)$ not only has no segments of the form ss, but any such segment contains an impossible in α subsegment of nearly-logarithmic length[13]. Let us cut off the tail of each binary string k according to some rule, say, the shortest one starting with 11. Let us fix a natural representation of all integers $i > 2$ by such tails $\widehat{i}$ and call i the *suffix* of k. Then $\alpha(k) = \langle \alpha[k], \mu(k) \rangle$. Here $\alpha[k]$ is k_i if $i \le |k|$, otherwise symbol #.[14] Let L_α be the set of all segments of α. It can be recognized in polynomial time.

Lemma 3 (Loop-catch) *Any string of the form ss, $|s|>2$, contains segments $y \notin L_\alpha$, $|y|=(\log|s|)^2+o(1)$.*

We say string $x = x_1 x_2 \dots x_k$ is *asymmetric* if it has a k bits segment of μ embedded in its digits (say, as $x_i \bmod 2$). For simplicity, we ignore other possible means of breaking symmetry.

Let $A^k(x)$ be a chain of k cellular automata A starting in the initial state with unchanging input string x. A^k *rejects* x if some of the automata enter a reject state. Language L is *t-recognized* by A if $A^{t(|y|)}$ rejects (1) all strings x with a segment $y \notin L$, within $t(|y|)$ steps, and (2) no strings x with all segments in L. For asynchronous self-stabilizing automata, requirement (1) extends to arbitrary starting configuration and to chains closed in a cycle; requirement (2) extends to the case when a tail/head of the chain is cut off during the computation.

Lemma 4 (Parallel Detection)
(i) Polynomial time languages of asymmetric strings are polynomially-recognizable by cellular automata.
(ii) Same for asynchronous self-stabilizing automata.

A.2 No-loop certificate: proofs

A *standard* (respectively, *shifted*) i-interval of μ is an interval of length 2^i whose starting (resp., center) position is divisible by 2^i. Any two adjacent standard i-intervals form either a standard or a shifted $(i+1)$-interval. We need to distinguish which. Since any standard interval is identical or complementary to an initial segment, it is convenient to represent a standard or shifted interval (of given length) by its first bit. Interestingly, such representation yields back the same sequence μ. No standard interval and one of any two adjacent shifted $(i+1)$-intervals consists of two identical standard i-intervals. So shifted $(i+1)$-intervals are recognizable, if the standard i-intervals are determined correctly. The standard intervals of μ naturally induce those of α. In arithmetic operations below, #, is treated as 0. For a standard i-interval x of α define $D(x) \stackrel{def}{=} \sum_{j \ge i : \widehat{j} < 2^i} 2^j$. $D(x)$ equals the distance $\bmod 2^j$, for $j : \widehat{j} < 2^i$, of the starting point of x in α from the start of α.

Proof of Lemma 3. Let x and y be two standard i-intervals, such that xy is a substring of α. Then $x[\widehat{i}] \equiv y[\widehat{i}]+1 \bmod 2$. Moreover, $D(y) = D(x)+2^i \pmod{2^j}$ for any $j > i, \widehat{j} < 2^i$. This condition is violated for any ss by picking (the smallest) i, j such that $i > \log \widehat{j}$, $j > \log|s|$. ■

Proof of Lemma 4 (i). Let L be a polynomial time language of asymmetric strings: there is a Turing Machine M testing $x \in L$ in time $|x|^c$, $c = O(1)$.

Let C be a chain of $|C|$ cellular automata containing some string s in the input fields. We need to show that $C(s)$ will detect substrings $x \notin L$, $|x|^c < |C|$, in time $|x|^{O(1)}$. Denote as μ_s the sequence (supposedly μ) embedded in the digits of s. Using the embedded $\mu_{(s)}$ sequence C organizes a hierarchy of (shifted) intervals. The possible corruption of μ_s is either detected or has no effect on the hierarchy construction.

Each shifted i-interval (for all i in increasing order) simulates $M(x)$ for all substrings x, $|x|^c < 2^i$. Since any standard $(i-1)$-interval is contained in some shifted i-interval, and a string of length $< 2^j$ is contained in some shifted or standard $(j+1)$-interval, any substring $x \notin L$ will be detected in time $|x|^{O(1)}$. ■

Proof of Lemma 4 (ii). Each standard interval is a trivial case of a tree. Thus, Prop. 2.1 provides self-stabilizing simulation of computation of any standard interval by asynchronous automata. Now, extension of Lemma 4(i) to cycles is trivial since no effect of the closing edge on the opposite side of the cycle can propagate to the short substring $y \notin L$ in time of its detection. Extension to the head/tale cut off is just by restarting computation on the intervals cut. The only problem in generalizing (i) to (ii) is that the intervals of all levels of the hierarchy must run simultaneously. (Otherwise, short strings $y \notin L$ cannot be detected

[13] Instead of $f(i) = i^2$ used below, let f be any such that $\sum_{i>0} 1/f(i) < \infty$. Then α can be redefined so that any string ss contains an impossible in α segment of length $f(\log|s|)$.

[14] Inclusion of μ in α is useful only for $<$ 40-bit segments. Also, $\mu(k)$ could be used instead of # if $i > |k|$ in $\alpha[k]$, but this complicates the coding and thus is skipped.

in time $|y|^{O(1)}$ since the relevant intervals must wait termination of computations of much larger intervals whose turn may be first in an adverse starting configuration.) So, we arrange the nested shifted intervals to simulate (repeatedly) their corresponding computations at the same time.

The main difficulty in such an arrangement is that the computations checking i-intervals for different i must use the same (constant) automata. So, we allocate (recursively) space for each i with density, say, $1/O(i^2)$. Below we consider the details of space allocation and its use for simultaneous checking of overlapping (nested) intervals.

Say, cell number k inside a standard i-interval belongs to *level* j if k's tail (as in definition of α, say the shortest starting with 11) is $\widehat{j}$ (representing j). If k contains no tail (no substring 11) the cell's level remains undetermined. If a cell belongs to level j in some i-interval, its level is the same in all larger intervals. Also, if an i-interval contains cells of level j then it also contains the cells of all levels $< j$.

Each level uses its cells to perform its computations (e.g. those described above). In addition, some levels (called *serving*) provide services to some higher levels. Each cell has a special bit marking whether the cell belongs to a serving level or not.

Let level i be serving level $j > i$. Then inside each shifted $i+1$-interval the first $\lceil \log i \rceil + 1$ of level j cells are specially marked as *address* cells. These cells are used to address an arbitrary location in the $i+1$-interval, which is then read into them by the level i. Using these functionalities (provided by lower levels) each level can perform its computations (which are described above).

Let i be marked as a serving level. Then in addition it must provide services to some higher levels (using half of its fields). Level $j > i$ is served by i if a shifted $i+1$-interval contains $> \log i$ cells of level j. The cells of the highest such j are marked by i as serving (for the higher levels). The levels j are served by i one after another introducing at most a polynomial slow-down in computations of levels j.[15] ■

A.3 SI for degree 2

Now we give details of SI for degree 2 networks. Its main part, *α-checker*, merely employs Lemma 4 to assure correctness of the no-loop certificate α. According to Lemma 3, α cannot loop and thus grows from a zero. After α-checker stops creating new zeros, SI, just simulates a TM (Prop. 2.1) checking and correcting the slope on each zero-zero interval. When ranks change, α-checker may replace the current α with another one, possibly growing from a different node.

Each node v has two internal α-checker fields (at most one of which may be empty): *old*, $v.\mathsf{a}_0$, and *new*, $v.\mathsf{a}_1$, each containing a digit of α, rank, and a pointer $\mathsf{p}(v.\mathsf{a}_i)$ to a $w.\mathsf{a}_i$ ($v \neq w \in \mathbf{E}(v)$ or $\mathsf{p}(v.\mathsf{a}_i) = v.\mathsf{a}_i$). These fields will be used by α-checker to guarantee zeros. Two copies are needed to build new α (keeping the old one until the new is finished) when dictated by (external) node rank changes. Define $v_*\mathsf{p}$ to point at the same node as v's oldest pointer ($\mathsf{p}(v.\mathsf{a}_0)$ if $v.\mathsf{a}_0$ is not empty, $\mathsf{p}(v.\mathsf{a}_1)$ otherwise). If $\mathsf{p}(v.\mathsf{a}_i) = v.\mathsf{a}_i$ then, its α digit is ignored ("correct") and $\mathbf{rank}(v.\mathsf{a}_i) \stackrel{def}{=} 0$.

α-checker 0-crashes (sets $\mathsf{p}(v.\mathsf{a}_0) = v.\mathsf{a}_0$, $\mathbf{rank}(v) = \mathbf{rank}(v.\mathsf{a}_0) = 0$, and $v.\mathsf{a}_1$ to empty) the nodes violating the following conditions: Any string of same rank α digits along $.\mathsf{a}_i$-pointers is in L_α (checked using Lemma 4). $\mathbf{rank}(v.\mathsf{a}_i) \geq \mathbf{rank}(\mathsf{p}(v.\mathsf{a}_i))$. $\mathsf{p}(v.\mathsf{a}_1)$ point at non-empty $.\mathsf{a}_1$ and $\mathsf{p}(v.\mathsf{a}_0)$ at non-empty $.\mathsf{a}_0$ or at $.\mathsf{a}_1$. If $\mathsf{p}(v.\mathsf{a}_0)$ points at a $w.\mathsf{a}_1$ then $v.\mathsf{a}_1$ must be empty, $\mathsf{p}(w.\mathsf{a}_0) \neq v.\mathsf{a}_i$, and if $w.\mathsf{a}_1$ is empty then $\mathsf{p}(v.\mathsf{a}_0)$ considered to be on $w.\mathsf{a}_0$ (it is reset to that when v acts).

Changes to $.\mathsf{a}_i$ violating the above are blocked (α-checker verifies the above condition before and after each change). Therefore, after polynomial time zeros exist and α-checker stops 0-crashing (LE create no crashes after a step, Lemma 1; no zeroing is used for the slope correction nor for α maintenance).

The new α grow and replace the old according to the following local rules (based on [Dij74]): If $v.\mathsf{a}_1$ is empty, $\mathsf{p}(v.\mathsf{a}_0) = u.\mathsf{a}_0$ for some u, $\mathsf{p}(x.\mathsf{a}_i) = v.\mathsf{a}_1$ for no x, either $w.\mathsf{a}_1$ is non-empty and $\mathbf{rank}(v) \geq \mathbf{rank}(w)$ for some $w \in \mathbf{E}(v)$ or $\mathbf{rank}(v) = 0$ and $\mathsf{p}(v.\mathsf{a}_0) = v.\mathsf{a}_0$ then $v.\mathsf{a}_1$ gets the pointer to $w.\mathsf{a}_1$ ($v.\mathsf{a}_1$ if zero), rank $\max\{\mathbf{rank}(v), \mathbf{rank}(w.\mathsf{a}_1)\}$ and the appropriate α digit. $v.\mathsf{a}_0$ is emptied when not blocked ($v.\mathsf{a}_1$ is non-empty and no pointers on $v.\mathsf{a}_0$). $v.\mathsf{a}_1$ is emptied into empty $v.\mathsf{a}_0$ when no neighbor w has empty $w.\mathsf{a}_1$, $\mathsf{p}(w.\mathsf{a}_0) \neq v.\mathsf{a}_1$, and $\mathbf{rank}(w) \leq \mathbf{rank}(v.\mathsf{a}_1)$ (and no $.\mathsf{a}_1$ point at $v.\mathsf{a}_1$). If $\mathbf{rank}(v) = \mathbf{rank}(v.\mathsf{a}_0)$, $v.\mathsf{a}_1$ is empty and v is closed, then v is opened.

Next, we show that each node is opened in polynomial time. Consider a zero-zero interval with endpoints z, z' of rank 0. Let w be the last with $w.\mathsf{a}_1$ leading to $z.\mathsf{a}_1$ (by $.\mathsf{a}_1$ pointers) and v be one further from z. If $\mathsf{p}(v.\mathsf{a}_0) = w.\mathsf{a}_1$ then the chain of $.\mathsf{a}_1$ leading to $z.\mathsf{a}_1$ shrinks each step ($w.\mathsf{a}_0$ is emptied and immediately gets $w.\mathsf{a}_1$ and, similarly, the for others). If $\mathsf{p}(v.\mathsf{a}_0) =$

[15] By reducing the number of levels served by each serving level this slow-down could be reduced. We avoid it for the sake of simplicity. As a side remark, with the above implementation there are $O(\log_* i)$ serving levels $< i$ for any i.

$u.\mathsf{a}_1$ for $u \neq w$ then within two steps $\mathsf{p}(v.\mathsf{a}_0) = u.\mathsf{a}_0$. If $\mathbf{rank}(v) \geq \mathbf{rank}(w)$, $v.\mathsf{a}_1$ is empty, $\mathsf{p}(v.\mathsf{a}_0) = u.\mathsf{a}_0$, then within a step there are no pointers on $v.\mathsf{a}_1$ and then $v.\mathsf{a}_1$ gets filled. Finally, if $\mathbf{rank}(v) < \mathbf{rank}(w)$ then v must have a path to another zero, which does not increase rank. If this path ceases to be rank non-increasing then w must change rank to $\leq \mathbf{rank}(v)$, w will not be able to increase rank after that until it is opened; v will get non-empty $v.\mathsf{a}_1$. If a path from v to z' stays rank non-increasing then by the above v gets non-empty $v.\mathsf{a}_1$ in linear time.

So, any node fills and empties $.\mathsf{a}_1$ within polynomial time. A node v may get $v.\mathsf{a}_1$ of $\mathbf{rank}(v.\mathsf{a}_1) \neq \mathbf{rank}(v)$ only if $\mathsf{p}(v.\mathsf{a}_1) = w.\mathsf{a}_1$ with $\mathbf{rank}(w.\mathsf{a}_1) \neq \mathbf{rank}(w)$. Thus the second time a closed node fills $.\mathsf{a}_1$ it has the rank of the node and thus the node will be opened when $.\mathsf{a}_1$ is emptied.

SI simulates a TM on each zero-zero interval verifying correctness of the slope on the interval. This is independent of the $.\mathsf{a}_i$ fields and is straight-forward. This TM can also provide a simpler way to coordinate opening of the nodes on the zero-zero interval when all slope and ranks of $.\mathsf{a}_i$ fields are correct.

A.4 Sequential *SI* for any degree

For general graphs, *SI* keeps a forest of forward $_*\mathsf{p}$ pointers. Acyclicity is assured by employing degree 2 *SI* (sec. A.3) on the DFS tour of each tree. Each $_*\mathsf{p}$-tree runs a TM simulating *SI* of sec. 3. Its BFS turns the forest into a BFS forest with a natural slope on it.

Each degree 2 graph (embeded along dfs) runs α-checker of sec. A.3, using $\mathbf{ord} \in \{\mathit{zero}, \mathit{low}, \mathit{med}, \mathit{high}\}$, $\mathit{zero} < \mathit{low} < \mathit{med} < \mathit{high}$, as ranks (*zero*, denoted by $v_*\mathsf{p} = v$, marks zeros). Height $v_*\mathsf{h}$ is defined as the variance of the $_*\mathsf{p}$-chain from v to the zero, for crashes using the distance by tree edges mod3 as the slope.

A TM (along the dfs tour) is maintained by each tree and provides all functionalities described below (α-checker not included). Two special marks, *server* and *client*, move around each dfs tour, each marking a (non-tree) edge. The path of $.\mathsf{a}_i$ pointers from each mark to zero (along the dfs tour; not passing through the other mark) must have rank *low* and the path connecting the two marks (say, directed from server to client) have rank *high*.

Whenever there is no client on the other end of its current edge, a server moves on to the next edge. A client waits to be served (by server on the other side of the edge) before moving on to the next edge.

When an edge vw is marked on both sides (by a client at v and a server at w) a new degree 2 graph is formed by joining the server's and client's cycles (possibly both on the same dfs tour). A version of the slope-checker, simulating *SI* of sec. 3, is run on this new cycle. First, for both trees simulate tr.1 on all tree edges for 0-crashing (no long tree edges are possible here). Then, if vw is not a long edge (with w at the lower end) both marks proceed to the respective next edges. (Pointers which are no longer forward can be corrected if both trees involved have found no long edges nor 0-crashed any nodes.)

Otherwise, if $v_*\mathsf{h} > w_*\mathsf{h}+1$ the tree pointer of v must be changed to w. This requires constructing new α. First, the server of the v's tree is pushed out of the subtree rooted in v, ignoring the client requests there. Then the dfs tour of the subtree of v (the path from v to its old zero) changes rank to *med*. When this is done, the pointer of v could be changed to w in a step (making the obvious adjustments at w and the old parent of v). But such a step of bfs can lower v's descendants in an undesirable way. So, before completing the pointer change the new slope needs to be computed for them, in a way simulating bfs on all of the subtree's edges. The height change of a node is equal to the height change of its parent if the parent was lower, one smaller if the parent used to be at the same height and two less if the parent was higher. Thus simulating bfs as above on all the subtree nodes we can complete the change of pointers (e.g. in a way similar to α-checker).

Consider a long edge vw with a lowest endpoint w. The server at w would never deny services to the client at v (both marking vw), since no ancestor of w has a long edge. Any server is moving along the dfs route spending only polynomial time on each edge and will thus in polynomial time appear on the other end of any waiting client. The server still may at that point deny services (if it is being pushed out from its own server's subtree). A client remaining in the same tree visits every tree node (and its edges). Every time a node switches the tree its distance to the zero (along tree edges; $\geq$height) is reduced. Therefore, after polynomial time vw will cease to be long.

A.5 Parallel *SI*

Finally, we sketch how the techniques introduced above are used to simulate *SI* of sec. 3. As opposed to the logarithmic version (where $v.\mathsf{h}$ was stored directly in a node) and to the sequential version (where $v_*\mathsf{h}$ was computed using the whole path from node to its zero) we will now compute $v_*\mathsf{h}$ using μ and α digits. Namely, $.\mathsf{a}_i$ fields will now contain (a modified, as described below) $\alpha(v_*\mathsf{h})$. Lemma 4 can be generalized to trees (rather than linear arrays) of automata.

We need to relax α: otherwise changing height will involve rebuilding the whole of the following α string (which may be linear in the size, rather than diameter). The relaxed α will still posses the quick loop-detecting properties, but will allow some "gaps", compared to α (which will allow to update only to the nearest allowed "gap", rather than the whole continuation of the old certificate).

A.5.1 Relaxed certificates

Intuitively, to construct relaxed certificates, $\alpha(k)$ is augmented to encode, in addition to the digits of k, also digits of its length, $\lceil \log k \rceil$. Then, a subsequence $\widetilde{\alpha}$ of (the augmented) α is a *relaxed certificate* if any two adjacent standard i-intervals in $\widetilde{\alpha}$ are adjacent in α or the first encodes a smaller length of the distance. In the definitions below, intuitively, an interval of the (augmented) α is complete (resp. semi-complete) if it encodes the distance (resp. the length of the distance). The formal definitions follow.

A standard i-interval x of α is *complete* if $x[\widehat{j}]=\#$ for some j, with the tail representation $\widehat{j} < 2^i$. [16]

Augment α with one more field β, encoding the size of the smallest complete interval, similarly to the way α encodes the distance from the start. Namely, $\beta[k] = r_i$, where i is the suffix of k (i.e. k ends with $\widehat{i}$), and r_i is the i-th bit of the smallest r, such that the standard r-interval containing k is complete. Also, similarly to the complete intervals, define a standard i-interval y to be *semi-complete* if it contains the complete encoding of the length of the complete interval: namely, if $y[\widehat{j}] = \#$ for some j with suffixless encoding $\widehat{j} < 2^i$. For a standard i-interval x of the augmented α define $L(x) \stackrel{def}{=} \sum_{j \geq i : \widehat{j} < 2^i} \beta_x[\widehat{j}] 2^j$, where $\beta_x[\widehat{j}]$ denotes the $\widehat{j}$-th β field of the interval x.

The (semi-)complete intervals are recognizable locally (first, recognize the corresponding i-interval of μ, and then check that for some j ($\widehat{j} < 2^i$), this interval contains # in the $\widehat{j}$-th position of α for completeness, in β for semi-completeness (counting from the beginning of the interval).

A subsequence $\widetilde{\alpha}$ of the augmented α is a *relaxed certificate* if for any standard i-interval x followed immediately by a standard i-interval y in $\widetilde{\alpha}$, if xy is a (standard or shifted) (i+1)-interval of the (augmented) α, or x is semi-complete and $L(x) < L(y)$ (or y is not semi-complete). Lemma 3 can be extended to relaxed certificates, and $L_{\widetilde{\alpha}}$ is polynomially-recognizable.

[16] Assume the suffixless representation of integers by tails to be monotone. Then $\widehat{k} < 2^i$ for all $k \leq j\,(< 2^i)$.