Flat Holonomies on Automata Networks*†

Please take the latest version from http://arXiv.org/abs/cs.DC/0512077

Gene Itkis‡ Leonid A. Levin‡

May 17, 2009

Abstract

We consider asynchronous networks of identical finite (independent of network's size or topology) automata. Our automata drive any network from *any* initial configuration of states, to a coherent one in which it can carry efficiently any computations implementable on synchronous properly initialized networks of the same size.

A useful data structure on such networks is a partial orientation of its edges. It needs to be flat, i.e., have null holonomy (no excess of up or down edges in any cycle). It also needs to be centered, i.e., have a unique node with no down edges.

There are (interdependent) self-stabilizing asynchronous finite automata protocols assuring flat centered orientation. Such protocols may vary in assorted efficiency parameters and it is desirable to have each replaceable with any alternative, responsible for a simple limited task. We describe an efficient reduction of any computational task to any such set of protocols compliant with our interface conditions.

1 Introduction

1.1 Dynamic Asynchronous Networks with Faults

The computing environment is rapidly evolving into a huge global network spanning scales from molecular to planetary and set to penetrate all aspects of life. It is interesting to investigate when such diverse complex unpredictable networks —including tiny and unreliable nodes— can organize themselves into a coherent computing environment.

Let us view networks as connected graphs of identical asynchronous finite automata and try to equip them with a self-organizing protocol. The automata have no information about the network, and even no room in their $O(1)$ memories to store, say, its size, time, etc. They run asynchronously with widely varying speeds. Each sees the states of its adjacent nodes but cannot know how many (if any) transitions they made between its own transitions. The networks must be *self-stabilizing*, i.e., recover a meaningful configuration if faults initialize their automata in any combination of states whatsoever.[1]

Such conditions and requirements may seem drastic, but stronger assumptions may be undesirable for the really ubiquitous networks that we came to expect. For instance, the popular assumption that each node grows in complexity with the size of the network, keeps some global information, and yet preserves reliable integrity, may become too restrictive (and is certainly inelegant).

So, which tasks and how efficiently can be solved by such networks? The network's distributed nature, unknown topology, asynchrony, dynamics and faults, etc., complicate this question. The computational power of any network with total memory n is in the obvious class Space(n). In fact, this trivial condition is sufficient as well.

---

*A preliminary version of this article appeared in [STACS-06].

†Supported in part by NSF grants CCR-0311411 and CCR-0311485.

‡Boston University, Department of Computer Science, 111 Cummington St., Boston, MA 02215.

[1]The faults are assumed *transient* i.e., self-stabilization is achieved after faulty transitions seize. Automata constant size and uniformity may help comparing neighbors and cutting edges to dissimilar ones. Absence of topology restrictions makes cutting-off persistently faulty nodes harmless.

1.2 Orientation and Computing

We consider protocols based on *orientation* for each directed edge (up, down, or horizontal) implemented by comparing Z_3 values held in nodes. It is a somewhat stretched transplantation to graphs of widely used geometric structures, *connections*, that map coordinate features between nearby points of smooth manifolds. Orientation is a simplest analog of such structures, comparing relative heights of adjacent nodes.

An important aspect of a connection is its *holonomy*, i.e., the composition over each circular path (often assumed contractible, though in graphs this restriction is mute). Connections are called *flat* if this holonomy is null (identity), for each cycle. For our orientations this means every cycle is *balanced*, i.e., has equal numbers of up and down edges.

Here is an example of utility of flat orientations. (Other types of connections on graphs might be beneficial for other problems, too.) Some networks deal with asynchrony by keeping in each node a step counter with equal or adjacent values in adjacent nodes. Nodes advance their counters only at local minima. For our model, such counters may be reduced mod 3 when no self-stabilization is required. The change of their values across edges induces orientation, obviously flat. Faulty configurations, however, can have inconsistent mod 3 counters with *vortices*, i.e., unbalanced (even unidirectional in extreme cases) cycles.

Flat orientations are especially useful when *centered*, i.e., having a unique node with no down edges. It then yields a BFS tree, maintaining which is known to self-stabilize many network management protocols.

Assuring these properties is the task of our automata. Their constant size combined with network's permissiveness, present steep challenges, require powerful symmetry-breaking tools, such as *Thue sequences* [Thu12] and others. These tools are highly interdependent: each can be disrupted by adversarial manipulation of others. This makes them hard to analyze, optimize, and implement.

Here we efficiently reduce these (and thus any other) tasks to several smaller problems; each can be solved completely independently as long as the protocols conform to a simple interface preventing them from disrupting each other. Such protocols may vary in assorted efficiency parameters, and it is desirable to have each replaceable with any alternative solving a simple limited task.

1.3 Maintaining Flat Centered Orientation

The task of assuring a non-centered flat orientation is easier in some aspects, e.g., it can be done deterministically. This is known to be impossible for the other task, centering an orientation. A fast randomized algorithm for it, using one byte per node, is given in [IL92]. The appendix there gives a collection of deterministic finite automata protocols that make orientation flat, running simultaneously in concert with each other and with the centering protocol.

In this paper we refer to three separate tasks: (1) rectify orientation on graphs spanned by forest of such trees, (2) center such an orientation merging the forest into a tree, and (3) fence vortices blocking centering process around them. Our main goal is to develop a protocol (4) *Shell* that (using no additional states) coordinates any (e.g., provided by an adversary) protocols performing these four tasks to assure that a centered orientation is verified and repaired if necessary, with the efficiency close to that of these supplied underlying task protocols. One more protocol (5) then efficiently reduces self-stabilization and synchronization of any computational task to assuring a centered orientation. The protocol (5) is described in Sec. 3. The tasks (1)–(3) are formally defined in Sec. 4, and the *Shell* protocol (4) is presented in Sec. 4.

1.4 Self-Stabilizing Protocols

The concept of *self-stabilizing* was pioneered by Dijkstra [Dij74] and has since been a topic of much research in distributed computation and other areas (see bibliography by T. Herman [Her]). Self-stabilization for typical tasks was widely believed unattainable unless nodes are not identical or grow in size (at least logarithmically) with the size of the network. (See, e.g., [M$^+$92] for discussion of undesirability of such assumptions.)

Logarithmic lower bounds for self-stabilizing leader election on rings [IJ90] (see also [DGS96]) reinforced this belief. However, such lower bounds depend on (often implicit) restrictions on accepted types of protocols: configurations with no potential leaders (tokens) must disappear in one step. Awerbuch, Itkis, and Ostrovsky [I$^+$92], gave randomized self-stabilizing protocols using $\lg\lg n$ space per edge for leader election, spanning tree, network reset, and other tasks. This was improved to constant space per node for all linear space tasks by Itkis in [I$^+$92], and by [IL92] (using hierarchical constructions similar to those used in other contexts in [Thu12, Ro71, G86]). These results were later modified in [AO94] to extend the scope of tasks solvable deterministically in $O(\log^* n)$ space per edge (beyond forest/orientation construction, for which algorithms of [IL92] were already deterministic).

There is extensive literature on self-stabilization and similar features in other contexts which we cannot review here. For instance, many difficult and elegant results on related issues were obtained for cellular automata (see, e.g., [G86]) on grids. However, the irregular nature of our networks presents different serious complications.

2 Models

Our *network* is based on a reflexive undirected (i.e., all edges have inverses) connected *communication graph* $G{=}(V,E)$ of n nodes, diameter d, and degree bound Δ. Nodes v are anonymous and labeled with *states* consisting of bits and pointers to adjacent nodes $w \in \mathbf{E}(v)$. Protocols are automata operating on functions of these states called *fields*. Their implementation specifies what changes of states actions on fields imply.

We avoid duplication when an edge carries pointers of several protocols as follows. The system call creates a ***hard*** pointer and sets a protocol's ***soft*** pointer to its name. Such soft pointer fields can be copied by other protocols. Hard pointers are removed when no soft pointers to them remain. A soft pointer can point at its source node; we then synonymously refer to it as absent or looping.

A ***link*** $[v, w]$ is the state of edge vw: a network obtained by renaming nodes v, w canonically and dropping all other nodes; pointers between v, w (incl. loops) are part of the link. Nodes ***act*** as automata changing their states based on the set (without multiplicity) of all incident links. Thus, a node's state transition may be conditioned on having (or not) neighbors in some state, but not on having five of them. When a node sets a hard pointer, it chooses a link, but not a specific (anonymous) neighbor connected by such a link. Some protocols may require this choice to be deterministic, e.g., using an ordering of edges. Thus, lemma 3.2 uses it on a tree to choose each child in turn for the TM simulation.

On a rooted tree with $\Delta = O(1)$, edges can be easily ordered by parents coloring them in Δ colors. Then, a general network N with a centered orientation allows a TM simulation by theorem 3.1. Such TM can use Δ^2 colors to color distinctly any nodes with common neighbors, thus ordering each node's edges in N. For non-constant Δ, cyclic ordering of node's edges needs to be provided by the model.[2]

2.1 Asynchrony

Asynchrony is modeled by *Adversary* selecting the next node to act: she adaptively determines a sequence of nodes with unlimited repetitions; the nodes act in this order. A network's (or protocol's P) ***step*** is the shortest time period since the end of the previous step within which each node acts (or P is called in it) at least once. By $\tau \succ s$ we denote that all of the step s occurs before the time instant τ. For simplicity, we assume that only one node acts at any time. Since node transitions depend only on its set of incident links, this is equivalent to allowing *Adversary* to activate simultaneously any independent set of nodes.

We could relax this model to ***full asynchrony*** allowing *Adversary* activate *any* set of nodes. This involves replacing each edge uv with a dummy node x and edges ux and xv. This change of the network affects only our structure fields protocols (assuring centered orientation: see Sec. 3.1), which tolerate any network. Node x is simulated by one of the endpoints, say u, chosen arbitrarily, e.g., at random. We call u *host* and x *satellite*; v, x — *buddies*. When activated by *Adversary*, a node first performs its own action and then acts for all its satellites. Thus, the dummy nodes never act simultaneously with their hosts.

To avoid simultaneous activation of buddies let each node (real or dummy) have a black or white color, flipped when the node acts (even if that action changes nothing else). A dummy node x acts only when its color is opposite to its buddy's; a real node v acts only when its and all its buddies' colors match. If a node does not act, in one step its buddies will have the color freeing it to act. Thus, at the cost of using a bit per edge, any structure protocol designed for our model can be run on a fully asynchronous network.

[2]For general undirected graphs, cyclic ordering of the edges for each node is equivalent to embedding the graph in a two-dimensional orientable manifold.

2.2 Faults

The ***faults*** are modeled by allowing *Adversary* to select the initial state of the whole network. This is a standard way of modeling the worst-case but transient, "catastrophic" faults. The same model applies to any changes in the network: since even a non-malicious local changes may cause major global change, we treat them as faults. After changes or faults are introduced by *Adversary*, the network takes some time to stabilize (see Sec. 3.1 for the precise definitions) — we assume that *Adversary* does not affect the transitions during the stabilization period, except by controlling the timing (see Sec. 2.1 above). Our protocols in this paper are all deterministic and make no assumptions about computational powers of *Adversary*. They may interact with or emulate other algorithms, deterministic or randomized. These other algorithms may impose their own restrictions on *Adversary*, which would be inherited by our simulations.

2.3 Orientation and Slope Bits

Edge ***orientation*** $\mathbf{dir}()$ of G maps each directed edge vw of G to $\mathbf{dir}(vw) \in \{0, \pm1\}$. The **rise** of a path $v_0 \dots v_k$ is $\sum_{i=0}^{k-1} \mathbf{dir}(v_i v_{i+1})$. We consider only orientations for which the *rise* of any cycle is $0 \pmod 3$. They have economical *representations*: Let each node v keep a ***slope bits*** field $v.\mathsf{h3}\in\{0,\pm1\}$ and define $\mathbf{dir}(vw)\stackrel{\text{def}}{=}-\mathbf{dir}(wv)\stackrel{\text{def}}{=}(w.\mathsf{h3}-v.\mathsf{h3} \bmod 3)\in\{0,\pm1\}$. We say that $w \in \mathbf{E}(v)$ is ***over*** v (and v is ***under*** w) if $\mathbf{dir}(vw)=+1$; directed edge vw points ***up*** and wv ***down***; define $\mathbf{up}(vw)\stackrel{\text{def}}{=}(\mathbf{dir}(vw)=+1)$. A path $v_0 \dots v_k$ is an ***up-path*** if v_{i+1} is over v_i for all $0{\le}i{<}k$. Cycles of 0 *rise* are called ***balanced***, others — ***vortices***.

A unique node with no down edges is called the ***center***. We will mark potential centers, calling them ***roots***. We call ***flat*** an orientation with roots, each with $\mathsf{h3} = -1$, only up edges, and *rise*≥ 0 outgoing paths. This implies no vortices and no up-paths[3] of $> d$ nodes, but is more restrictive than in the Introduction (Sec. 1). A flat orientation with a center is called ***centered***.

2.4 Tree-CA Time and TM Reversals

We characterize in usual complexity terms the computational power of asynchronous dynamic networks G in two steps. First we express it in terms of Cellular Automata H on G-spanning trees (*tree-CA*). We treat H as a special case of our networks when they are trees initialized in a blank state and acting synchronously. H holds the network topology as adjacency lists l_v (say, by the dfs numbering of the tree) of its nodes v. l_v are held in read-only ***input registers***; v have access to one bit of l_v, rotated synchronously by the root.

Once its flat orientation stabilizes, our network can simulate tree-CA (subsection 3.2). Tree-CA are simpler than our networks, but still have significant variability depending on the topology of the trees. To avoid this variability, we further compare them in computational power to Turing Machines (TM). Tree-CA can simulate TMs and vice versa (subsection 3.2). The efficiency of this mutual simulation seems best expressed using the number of ***reversals*** i.e., changes of the TM head direction as (parallel) time complexity. When using this measure [Tra64, Bar65], we refer to TM as *reversal TM (rTM).*

Our rTM has read-write work and output tapes W, O of size $\|W\| = \|O\| = n$, and a read-only input tape I. For simplicity we assume rTM's heads turn only when the work head is at the end of its tape. The bits of tree-CA input registers are stored on rTM's input tape at intervals $2n$, so that when the work-tape head is in cell i, the input-tape head reads a bit of the i's register.

Ignoring d, Δ time factors, tree-CA on any tree have the same computing power as rTM with the same space and time, thus exceeding power of sequential RAM. rTM can simulate RAM fast but can also, say, flip all bits in one sweep, which takes $\theta(n)$ RAM time. Variant connectivity gives some networks greater power of parallelism than others. For instance, tree-CA take nearly linear time to simulate sorting networks, while the latter given read-only access to the adjacency list of any other network, can simulate it (or PRAM) with polylog overhead.

[3] Such paths determine delays in many applications, but higher limits often suffice. Many algorithms modify orientation gradually, changing *rise* of any path by at most 1 at a time. Then the *rise* of any cycle (being a multiple of 3) stays constant. This limits the cumulative *rise* change of any path to $\pm 2d$. Thus, the maximum node-length of up-paths can vary with time by at most a $2d$ factor.

3 Solving Any Task with Centered Orientation

Consider an rTM algorithm $T_n(x)$ that computes a function $t_n(x)$ when initialized on a working tape of size n with x on the input tape. T, t are called ***constructible*** if T runs in (reversal) time $O(t)$ and space $O(n)$. The running time of any algorithm T is constructible since T can be modified to count and output its time.

We need to tighten this condition slightly to assure the time bound even when T is initialized in maliciously chosen configurations. We call algorithm T, and the function $t_n(x) > \lg n$ it computes, ***strictly constructible*** if for some $c \in (0,1)$, T runs in space $O(n/|\lg_c n|)$ with $O(t^c)$ expected reversals. Most functions t used as time bounds take for their computation significantly (usually exponentially) less time and space than $t_n(x)$ steps and n cells. Thus, the overheads of strict constructibility are rarely an issue.

Let q be an input-output relation on pairs $\langle x, y\rangle$ of questions x and "correct answers" $y \in q_x$. With a strictly constructible time bound $t_n(x)$ it forms a ***task*** Γ if there exist a pair $\langle \Lambda, \Phi\rangle$ of probabilistic algorithms: Checker (needed only if $\|q_x\| > 1$) and Solver, running in space $\|y\|$ and expected time $t_n(x)$ such that

- $\Lambda_n(x, y)$ never rejects any $y \in q_x$, but with probability $> 1/2$ rejects every $y \notin q_x$;
- $\Phi_n(x)$ with probability $> 1/2$ computes $y \in q_x$.

Our goal is for any task (specified for a faultless and synchronous computational model such as rTM) to produce a protocol running the task in the tough distributed environment where *Adversary* controls the timing and the initial state of the system. We separate this job into two: First, we assume that some special ***structure protocols*** generate a centered orientation and stabilize, i.e., the orientation stops changing. Section 3 and its Theorem 3.1 discuss how to achieve our goal after that. The remainder of the paper starting with Sec. 4 describes the structure protocols, which run in the special *structure* fields.

3.1 Self-Stabilization

Let each processor (node) in the network G have read-only ***input*** field, and read/write ***work, output***, and ***structure*** fields. A ***configuration*** at time instant τ is a quintuple $\langle G, I, O_\tau, W_\tau, S_\tau\rangle$, where functions $I, O_\tau, W_\tau, S_\tau$ on V represent the input, output, work and structure fields respectively. The structure protocols serve to maintain the centered orientation. They run in S_τ, are independent of the task and computation running in W_τ, O_τ, and affect it only via setting the orientation fields of S_τ which the computation can read.

Let q be a set of correct i/o configurations $\langle (G,I), O\rangle$, and $\Gamma = \langle T, q\rangle$ be a corresponding task. A protocol ***solves*** Γ ***with self-stabilization in*** s ***steps*** if starting from any initial configuration, for any time $\tau \succ s$ the configuration $\langle (G, I), O_\tau\rangle \in q$. For randomized protocols we measure the expected stabilization time. Our protocols do not halt, but after stabilization their output is independent of the subsequent coin-flips. (For synchronized protocols stabilization could also include repetition of the configuration.)

Protocols, which accept (potentially incorrect $\langle (G,I), O'\rangle \notin q$) halting configurations, cannot be self-stabilizing: the network put by *Adversary* in an incorrect halted configuration cannot correct itself. Our protocols for Γ repeatedly emulate checker Λ, invoking Φ when Λ rejects an incorrect configuration. We use here the Las Vegas property of (properly initialized) Λ: it never rejects a good configuration. *Adversary* may still start the network in a bad configuration from which neither Φ nor Λ recover within the desired time. To handle this, we use the self-stabilizing timer T constructed in Lemma 3.1.

Remark 3.1 (Dynamic Properties) *For simplicity, we focus on "static" problems. However, the dynamic behavior of protocols is often of interest as well. We note that many temporal properties can be achieved by creating (with self-stabilization) a static configuration that, once correctly established, allows regular algorithms (without self-stabilization or asynchrony resistance) to assure the desired behavior.*

Theorem 3.1 *Any task Γ can be solved on any asynchronous networks G with (unchanging) centered orientation in their S-fields by protocols self-stabilizing in $T(G, I)O(d\Delta \lg n)$ steps.*

For a proof we define a ***stably constructible*** rTM $T_n(x)$ (or ***timer***) as one that starting from *any* configuration on n-cell work tape, stabilizes with $O(T_n(x))$ expected time.

Lemma 3.1 *Any strictly constructible function t can be computed by a stably constructible algorithm.*

When $T_n(x)$ is a timer, any task can be self-stabilized. M keeps two counters t, r and runs T repeatedly. Whenever T halts, its output overwrites t. Each step, r is decremented if $r \in [1, t]$. Otherwise, r is reset to t and M runs Λ, properly initialized. If Λ rejects, M runs Φ. If outputs of Φ are unique, no Λ is needed: Φ is run always but its rewriting correct outputs makes no changes and does not disrupt the stabilization.

Proof of Lemma 3.1 Let $C = \lceil 1/(1-c) \rceil$; we round c to $1 - 1/C$. First, we set a $\lceil \lg n \rceil$ steps rTM timer. It sweeps the tape, each time marking every second unmarked cell. When all are marked, it unmarks the tape, and restarts. With it, we stabilize the following $O(k)$ steps task. It computes $k = \lceil \lg n - \lg(C \lg n) \rceil$ similarly to the above timer, and by k merges divides the tape into numbered segments s_i of length 2^k (s_0 may be shorter), each keeping a binary counter r_i bounded by t_i with $\|t_1\| = C$, $\|t_{i+1}\| = \lfloor \|t_i\|/c \rfloor \approx c^{-i}$).

In each s_i, rTM runs $T(x)$ (iterated to error probability $< 1/3k$ if randomized), in parallel. The i-th run goes for t_i steps and restarts from the blank state. If it halts, all other runs are restarted, too. Thus, if $T(x)$ takes $T_x \in (t_{i-1}, t_i]$ steps, then starting from any configuration, within $t_i < T_x^{1/c} < T(x)$ steps the i-th run restarts from blank state and halts in $< T(x)$ expected time. ∎

3.2 Tree-CA, rTM, and Network Simulations

In this section, we consider how tree-CA H and an rTM M can simulate each other. Let H have n nodes and M have $2n$ cells, numbered from left to right. We map each node x of H to two cells of M, denoted $x_($ and $x_)$ reflecting the two visit times of dfs traversal of H. Let input tape bits M reads when its work head is at nodes $x_(, x_)$ and bits in the input register of x reflect each other. Let functions $h, g_), g_($ map the tape characters of M to the automaton states of H and vice versa. We say a machine A *simulates* B *with overhead* t if after any number i of steps (or sweeps) of B and ti steps of A, the state of each cell (or node) of B is determined by the function h or g applied to the corresponding node of A.

Lemma 3.2 *Any tree-CA H (diameter d, degree Δ) and rTM M with matching inputs, can simulate each other: H with overhead $O(d\Delta)$ and M with $O(d)$.*

Proof: **H simulating M.** The automata nodes x of each depth in turn, starting from the leaves, compute the transition function f_x. This f_x depends on the current states and inputs of the subtree t_x of x and its descendants. It maps each state in which M may enter t_x from the parent of x (sweeping the tape along the dfs pass of H) to the state in which it would exit back to the parent. Once f_y is computed for each child y of x, the new states of $x_), x_($ and f_x are computed in $O(\Delta)$ more steps. Since the depth of the tree is d, it takes $O(d\Delta)$ to compute f_{root}, and thus to simulate one sweep of M work tape.

M simulating H. Each node x of H corresponds to a pair $x_($, $x_)$ of matching parentheses enclosing images of all its descendants (in t_x). On each sweep M passes the information between matching parentheses of certain depth. Nodes x at this depth are marked as *serve*, their descendants as *done*, and their ancestors as *wait*. When the root is *done*, all marks are turned to *wait* and M starts simulating the next step of H (from the leaves). When $x_($ and $x_)$ *wait* and their children *serve*, M serves $x_(, x_)$ as follows.

The next sweep carries the state of x to its children allowing them to finish their current transition and enter *done*. The same sweep gathers information from the children of x for the transition of x and carries it to $x_)$. The return sweep brings this information to $x_($; at this point, $x_(, x_)$ go into *serve* state — only the parent of x information is needed to complete the transaction of x.

M keeps two counters: for the input register place all automata of H read at this simulation cycle, and for the segment of input tape M reads at this sweep. M reads its input when the counters match. ∎

Proof of Theorem 3.1 A centered orientation on G yields a spanning bfs tree via its up edges. Consider a tree-CA H on it. It can be synchronized by keeping a second orientation, incrementing its slope bits and making a step in each node with no tree-neighbors under it. H in turn emulates an rTM M. We also need G to simulate the rotating registers of H carrying addresses of their G-neighbors.

The vertices are numbered linearly on the tape of M covered with counters, each with the number of its first vertex. Such counters are initialized in $O(\lg n)$ time similarly to marking the intervals in Lemma 3.1 proof. The root keeps a (rotating) place i and all points display the i-th digit of their numbers, giving access to it to all network neighbors. An adjacency list look-up can thus be simulated in $O(d\Delta \lg n)$. ∎

4 Assuring Centered Orientation: Problem Decomposition

The protocols in Theorem 3.1, use centered orientation (in h_3 fields, Sec. 2.3). The rest of the paper reduces assuring such an orientation to three separate tasks of: orientation Rectifier **R**, Leader Elector **LE**, and Fence **F** blocking **LE** around vortices. This section presents these tasks in terms of interfaces (read/write permissions for fields a protocol P shares with its ***environment*** $\mathcal{E}_\mathcal{P}$) and commitments (with time parameters $\mathsf{t}_\mathbf{R}, \mathsf{t}_\mathbf{LE}, \mathsf{t}_\mathbf{F}$). Any protocols complying with these *contracts* will work for our reduction, given below as the **Shell** protocol **Sh**. **Sh** uses only one bit $\mathsf{b}_\mathbf{F}$ and one pointer p_b (it also reads pointer p_l).[4]

Legality, Guard, and Crashing. *Adversary* initiates the network with arbitrary links, possibly "abnormal," disruptive for P. Correcting them might be hard for P: it is restricted by the interface and acts at one node at a time, affecting all incident links, not just abnormal ones. Let P come with a list of ***P-legal*** links; v is ***P-legal*** if all links exiting it are or if v to $\mathsf{on}\!\uparrow$ (defined below). Any activated v invokes a function *guard* $\mathcal{G}$, with the list of illegal links and access to all fields. It ***crashes*** illegal v into $\mathsf{on}\!\uparrow$, and does nothing else. P-legality of nodes and in-links must be preserved by crash and any actions P makes or permits to $\mathcal{E}_\mathcal{P}$.

Shell fields. $\mathcal{G}, \mathbf{R}$ (and only they) create roots – potential centers of the orientation. **LE** "uproots" them and, in non-roots, calls Float which, with no edges to roots or down, increments h_3. Eventually the orientation has a center led to by all down paths. Uprooting creates non-root local minima, and thus, down-paths not leading to roots. To guide to roots, **LE** keeps ***lead*** pointers $v.p_l{=}\vec{v}$; p_l ***loops*** ($\vec{r}{=}r$) in roots, cutting off pointer chains. Invoking **LE** at v, **Sh** copies $v.p_l$ to the ***backup*** $v.p_b$ (to help other protocols adjust if **LE** changes p_l). **Sh** initiates **F** on a p_l-tree by turning on its root's ***fence bit*** or ***phase*** ($r.\mathsf{b}_\mathbf{F}{\leftarrow}1$); **F** exits turning it off ($r.\mathsf{b}_\mathbf{F}{\leftarrow}0$; only **F** can turn the roots off).[5]

Notation. $\mathcal{L}_l$ (***stub*** $\ddagger$), $\mathcal{L}_b$: p-loop predicates; $v.p_{bl}$: $(v.p_b).p_l$; $\mathcal{L}_{bl}$: $v.p_{bl}{=}v$, etc. Adjacent stubs are ***lock***s, isolated – ***roots***. $\mathcal{L}_i$: $i{=}\mathcal{L}_b{+}\mathcal{L}_l{\in}\{0,1,2\}$. $\circ$: $\mathcal{L}_2$; ***single*** $\downarrow$: $\mathcal{L}_1\&\mathcal{L}_b$; ***reset*** $\uparrow$: $\mathcal{L}_1\&\mathcal{L}_l$. ***Duplex*** $\downdownarrows$ ($\mathcal{L}_0$) are ***double*** $\Downarrow$ if $v.p_b{=}\vec{v}$, else ***hook*** $\downarrow\!\upharpoonright$ if $\mathsf{off}\,\&\,\mathcal{L}_{bl}\,\&\,v.p_b{\notin}\Downarrow$, ***split*** $\curlywedge$ otherwise. ***Ground***: root or $\mathcal{L}_{bl}$ split. We denote **Sh** states by $\mathsf{b}_\mathbf{F}$ and pointer pattern (e.g., $\mathsf{on}_\circ, \mathsf{off}\!\downarrow$).

Height. ***Senior*** pointer $v.p_B$ loops in ground, is $v.p_b$ in other splits, $\vec{v}$ otherwise. The ***height*** $h(v)$ of v becomes undefined ($\perp$) when v is lock or crashes, and remains so until non-lock v changes **Sh** field(s). Otherwise $h(v)\overset{\text{def}}{=}v.\mathsf{h}_3$ in ground v. For other $v{\neq}v.p_B{=}w$, $h(v)$ is $h(w){+}\mathbf{dir}(wv)$, retaining its previous value if $h(w){=}\perp$. A directed edge vw becomes ***bound*** when w or its p_l-descendant changes $\mathsf{b}_\mathbf{F}{\leftarrow}1$, or the senior ancestor root of v or w changes between $\mathsf{on}_\circ$ and $\mathsf{on}\!\uparrow$. It reverts to ***unbound*** when v crashes. Around vortices *rise* varies with paths and edge ends may differ by >1 in height; such edges are called ***rips***.

Symmetry breaking. **R** (with minimal help from **F**) maintains a hierarchic structure on trees to enable initiation of parallel **R** protocols. It is kept via sign bits $\lambda(k)$ of $v.\mathsf{h}_3{=}\pm 0$, where $k{=}h(v)/3{=}2^i(4j{+}s)$, $s{=}\pm1$ and $\lambda(k){=}\mathrm{sgn}(s)$.[6] As an exception, we set $\lambda(k{+}1)$ to $-$, marking "round" $k{=}2^i(4^c{+}j^2), j{<}2^c$ with an otherwise impossible ***mark*** pattern $-{+}{-}{+}$. Here c is a constant that depends on the one in the commitment (**LE**.ht) below. Any segment of λ with two marked heights determines them uniquely. Thus, **R** can use the slope bits h_3 to quickly detect rips even when the senior chain is much larger than the height.

4.1 Protocols

Interface permissions. Read restrictions serve only to help reader's focus; write restrictions apply only to the ***shared fields*** ($\mathsf{h}_3, p_l, p_b, \mathsf{b}_\mathbf{F}$). $\mathcal{E}$ of each protocol can do all actions of **Sh**, and (when **Sh** calls other

[4]The tasks of **R** and **F** correspond roughly to the two functions of *SI* in [IL92] – initiating a flat slope and keeping nodes open for **LE**. While [IL92] protocols comply with our contracts, they had other interdependences and were not designed to take full advantage of the efficiencies allowed by the separation provided here by **Sh** and contracts. *SI* was concerned only with $n^{O(1)}$ time-bounds, while here our **Sh** preserves the efficiency up to factors $d^{O(1)}$, possibly exponentially smaller than the number of nodes n. Our present **Sh**, **F**, and (sketched in the appendix) **R** adjust *SI* tasks to the new opportunities.

[5]The fence bit $\mathsf{b}_\mathbf{F}$ is used to pass control between **F** and **Sh** analogously to the control bit in [IL92].

[6]This sequence λ is based on one used (implicitly) in [Ro71], and discussed in [Le05]. [IL92] uses instead $\mu(k)$ (based on [Thu12]) defined as "$-$" if binary encoding of k has an odd >1 number of 1s, or "$-$" otherwise.

protocols P) those listed below as permitted to P. v is ***ready*** if $\vec{v}=v$ or $v.\mathsf{b}_{\mathbf{F}}\neq\vec{v}.\mathsf{b}_{\mathbf{F}}$, or $v\in$⇓, $\vec{v}\in$↑. $\mathbf{R},\mathcal{G}$ can crash any v. Otherwise, shared fields change only in ready v with no ready p_l-child, and **R** can change only locks (not to off with p_l-children). **F** changes only $\mathsf{b}_{\mathbf{F}}, p_b$ in roots and $\mathsf{h}_3=\pm 0$ signs. **R** can ***open*** lock v into on↓ with $v.p_b\in$on↑ under v, all down and no up edges of v going to stubs. **R** can decrement h_3 of locks with no up edges to non-stub, and change ± 0 sign. Only **R** can set off↑. **LE** reads $\mathbf{dir}(), p_l$, calls Float and moves $v.p_l$ (to $\neq v$); it idles in v, w if $\vec{v}=v$ and $\mathbf{dir}(vw)\neq 1$.

Shell. **Sh** starts by changing $\mathsf{off}_\circ$ to $\mathsf{on}_\circ$, and invoking $\mathbf{F},\mathbf{R},\mathcal{G}$; locks with ⇓-children change to on↑. Invoked in other (ready) non-locks, **Sh** does the following.

Split: v invokes $\mathbf{LE},\mathbf{F},\mathbf{R}$ if v (1) is $\mathsf{off}_\circ$, or on↓ with $\mathcal{L}_b(\vec{v})$, has (2) a ⇓ or no child, (3) no split p_b-child, and (4) no ↓child. Before this, **Sh** sets p_b to p_l or, in root, to a ⇓child, if any. Uprooted childless v turns ⇓.

Merge: Activated as ⅄, v merges (1) into ⇓ if $\vec{v}\in$off↓ and v has ↓ or no child, (2) into ↓ if $\vec{v}$ is ⇊ and (a) $v\in$on has a ↓ or no child, or (b) v has off children, all ⅄ or ⥯. ⇓ merges into ↓ if $\vec{v}$ is ↑ or on⇊.

Phase Wave: Then **Sh** sets $v.\mathsf{b}_{\mathbf{F}}\leftarrow\vec{v}.\mathsf{b}_{\mathbf{F}}$, changing ⥯ to ↓, and ↓ with a child and a ⇊ parent, to ⥯.

Commitments. After the first step (when $\mathcal{G}$'s crashes stop) under the above Interface and Shell:

- **(LE.ht):** **LE** assures a segment of *rise* $c\cdot m$, $c=\theta(1)$ in any m-node p_l-chain.
- **(F.cln):** **F** assures that no v with $v.\mathsf{b}_{\mathbf{F}}=0\neq\vec{v}.\mathsf{b}_{\mathbf{F}}$ has a $\mathsf{b}_{\mathbf{F}}=0$ p_l-ancestor.
- **(F.sgn):** **F** sets the sign of h_3 to $\lambda(h(v)/3)$ in (ready) v with a bound in-edge and $\mathsf{h}_3=\pm 0$.
- **(F.rip):** **F** assures that senior chains from bound rips do not change.
- **(R.stb):** With the above commitments, **R** ***stabilizes*** in $\mathsf{t}_{\mathbf{R}}$ steps: crashes stop, orientation is flat.
- **(F.off):** **F** turns each root off every $\mathsf{t}_{\mathbf{F}}$ steps after **R** stabilization.
- **(LE.ct):** **LE** centers orientation within expected $\mathsf{t}_{\mathbf{LE}}$ **LE**-steps after **R** stabilization.

4.2 Shell Performance

A non-lock is ***low*** if it has only ↓ and ⇟ ancestors (incl. self), ***high*** otherwise; a high with a low parent is ***border***. Only on↓ occurs in both high and low (but not border). A node becomes high (border) only as a result of invoking **LE** in leaves of low. A root, after invoking **LE** (unless uprooted) resets its tree to low by passing through on↑, and a new cycle of **LE** calls starts. Intuitively, **F** waits for the whole p_l-tree to turn on, checks it for rips (more precisely, vu such that the root-root path against p_l pointers, across vu, and then along senior chain, has non-0 variance), and, if none, turns the tree root off (then **Sh** propagates off through the tree). Turning off, double children of a split become single, so the split merges at the next off —after completing a full **F** cycle with its checks. However, a split v merges ***prematurely*** if it has no children (when turning off) or if it has only split children and $\vec{v}\in$on⇊ (thus, e.g., as a p_l-chain of splits turns on, the alternating ones merge prematurely; the remaining splits will merge upon the next off wave). Uprooting, r, if childless, instantly merges into its new tree; if with a double child w, remains ground (but now a split).

We show that centered orientation will be assured by any protocols that satisfy the above commitments. For the rest of the subsection assume that **R** has stabilized (**R**.stb): the orientation is flat (incl. has roots, no locks), **R** no longer changes any shared fields (and thus can be ignored). Then any p_l-chain is at most $O(d)$: the orientation flatness bounds *rise* by $O(d)$, and (**LE**.ht) extends this bound to the length of p_l-chains. For every root r, **F** changes $\mathsf{on}_\circ$ to $\mathsf{off}_\circ$ within $\mathsf{t}_{\mathbf{F}}$ steps (**F**.off), and then (unless r uproots) **Sh** changes it back to $\mathsf{on}_\circ$ in one more step (after all its p_l-children had a chance to copy $r.\mathsf{b}_{\mathbf{F}}$). Assume $\mathsf{t}_{\mathbf{F}}=\Omega(d)$ (otherwise we may need to replace $\mathsf{t}_{\mathbf{F}}$ with $\mathsf{t}_{\mathbf{F}}+d$ below). A node v is a ***switch*** if $v.\mathsf{b}_{\mathbf{F}}>(\vec{v}).\mathsf{b}_{\mathbf{F}}$.

For any v, $v.\mathsf{b}_{\mathbf{F}}=1$ within $O(d)$ steps. Indeed, let $v.\mathsf{b}_{\mathbf{F}}=0$. (**F**.cln) assures any on p_l-child of v has no off child, in a step all children of v are off. The maximal off p_l-chain from v gets shorter within each step.

For any v, $v.\mathsf{b}_{\mathbf{F}}=0$ within $O(d\mathsf{t}_{\mathbf{F}})$ steps. Indeed, a low on v changes to off or high within $O(\mathsf{t}_{\mathbf{F}})$ steps: its root is turned off or uproots (making v high) within $\mathsf{t}_{\mathbf{F}}$ (**F**.off); if its root is off, the on p_l-chain from low v shrinks ($O(d)$ times) within a step till v either splits or changes to off. After the initial $O(d)$ steps a high node does not invoke **LE** (an on↓ with an off⇊ parent changes to off). Then for a high v consider the maximal high on p_l-chain to a split. This chain can only shrink if the split changes to off (and then within $O(d)$ so does v). Within $O(\mathsf{t}_{\mathbf{F}})$ steps the chain either grows (at most $O(d)$ times) or v changes phase: its nearest low ancestor becomes off or high within $O(\mathsf{t}_{\mathbf{F}})$, either becoming a split (increasing the chain), or off⇓ (and then the on p_L-chain from v shrinks each step). Thus, $v.\mathsf{b}_{\mathbf{F}}=0$ within $O(d\mathsf{t}_{\mathbf{F}})$ steps.

A split v can change $v.\mathsf{b}_{\mathbf{F}} \leq 3$ times without merging, thus v merges in $O(d\mathtt{t}_{\mathbf{F}})$. Indeed, when v changes to $\mathsf{on}\downdownarrows$ it looses its double children (or merges). Then it merges by the next change to off.

A low v invokes **LE** *within $O(d^2\mathtt{t}_{\mathbf{F}})$.* Indeed, any low leaf looses its split p_b-children in $O(d\mathtt{t}_{\mathbf{F}})$ (similarly, if it is a root its existing split children merge into $\Downarrow$), and then invokes **LE** the next time it is a switch (or $\mathsf{on}_{\circ}$ with only $\mathsf{on}\Downarrow$ children). The depth of the low node (sub)tree (of v) can be so reduced $O(d)$ times.

Lemma 4.1 *Any node invokes* **LE** *within $O(d^3\mathtt{t}_{\mathbf{F}})$ steps.*

Indeed, consider a high v and the shortest p_l-chain from v to a border, split or ground w (possibly $=v$). Such a chain cannot shrink without v invoking **LE**: new grounds are not created any more (except when a childless root floats possibly making its new parent a ground) and new splits are created with only border children. Moreover, within $O(d^2\mathtt{t}_{\mathbf{F}})$ steps the chain grows or v invokes **LE**: If w is a root, then v is low (and invokes **LE** within $O(d^2\mathtt{t}_{\mathbf{F}})$); otherwise, if w is a split, it merges in $O(d\mathtt{t}_{\mathbf{F}})$; and if w is a double, then in $O(d^2\mathtt{t}_{\mathbf{F}})$ $\vec{w}$ invokes **LE** and w changes to single $O(d\mathtt{t}_{\mathbf{F}})$ steps later. Since this chain can grow only $O(d)$ times v will invoke **LE** within $O(d^3\mathtt{t}_{\mathbf{F}})$ steps. ■

Since **LE** interface fields are not affected by any other protocols, this lemma implies ***prompt*** (polynomial in the network diameter d and degree Δ) centralization:

Theorem 4.1 (Main) *Given any contract abiding protocols* $\mathbf{LE}, \mathbf{R}, \mathbf{F}$*, our Shell* **Sh** *assures centered orientation within expected* $\mathtt{t}_{\mathbf{R}}(\Delta, d) + O(d^3\mathtt{t}_{\mathbf{F}}\mathtt{t}_{\mathbf{LE}})$ *steps.*

5 Fence F

Intuitively, the main function of **F** is to prevent changes of senior chains from rips. Only locks and splits may change their senior pointers, and thus their and their descendants' senior chains (and heights).

Call v ***hanging*** if the p_l-chain from v has a long p_l.[7] An ***apex*** is a low v with no low children; when v becomes a switch it might split (or float). An on-apex v is ***loose*** if it has no p_l-children: it can split and then merge prematurely (without completing a full **F** cycle, see below). To assure (**F**.rip), **F** needs to check that its tree has no incident rips (including p_l), but such a check is unreliable if a neighbor v is (1) hanging; (2) childless low with a hanging neighbor or a long edge; (3) childless low with a childless low neighbor u and a long edge uw to a low w.[8] Such v can change height creating rips for its (possibly already checked) neighbors. So, in addition to rip-checking incident edges, **F** must assure that before getting an off p_l-ancestor, (1) its high neighbors will check that they are not hanging, and (2) its childless low neighbors v will rip-check their edges and in turn assure that their childless low neighbors u have no rips uw to a low w. This requires two "milestones" in the high nodes and three for the childless low nodes. So, next we describe the **F** cycle which achieves these "milestones"; then we describe the rip-checking and exiting from locks.

5.1 F cycle

F cycle is initiated on a p_l-tree from its root by switching to on ("registering" p_l-pointers forming the tree; a p_l pointer joining the tree after this registration will participate only in the subsequent **F** cycle). Unless specified otherwise, the parents and children below refer only to these (registered p_l-) tree edges.

Transitions. The **F**-cycle consists of two phases (0 and 1), each with three states: start, active, done. Intuitively, the goal of phase-1 is to provide assurance (to the neighbors) of height preservation, while phase-0 is focused on assuring no rips (for its own nodes). In a regular **F** cycle phase-0 is run once (following off wave), while phase-1 is potentially re-cycled repeatedly (until the next off), from an unregistered split.

In high nodes the states function similarly to the classical children game of fire-water-hay: with fire (start, propagating up: from parent to children) consuming hay (done), but put out by water (active, propagating down: from children, when all active, to parent), which in turn is absorbed by hay (done, propagating down, similarly to active).

[7] An alternative more precise definition is possible: the p_l-chain from v to the long edge contains no splits with double children and no nodes that were ever off.

[8] In the last case, u can split to w, while w is on; changing w to off will result in u prematurely merging into a double (changing its height and making vu long); then v can split to u while u is still off, and have another neighbor split to v; then changing u to on will result in the premature merge of v, changing its height.

In low nodes the transitions are slightly more complex: there start, done and active-0 propagate in the same directions as in high, but active-1 propagates from parent to children. More specifically, done-0 in low nodes is delayed while active-0 (which enters a low node only when all its high children enter done-0) propagates to the root turning into active-1 signal propagating back towards done-0. Then done-0 propagates on low (replacing active-1) towards root. Upon reaching the root, done changes into start-1, which propagates up replacing done-0. Similar to phase-0, a low start-1 does not change until its high children are all done-1, but here it changes directly to done-1 which proceeds towards the root (consuming start-1 parents). A root with all children done-1 changes to off, signaling that **F** is finished on this tree.

If a node v in done-1 (and all children in done-1) splits or uproots, then it recycles phase-1 on its subtree until changing to off: done-1 with low (also done-1) parent changes to start-1. Thus, intuitively, start propagates always from the parent to the children; and done — from the children (when all are done) to the parent; active-0 propagates similarly to done, while active-1—towards border: as an echo (preceding done-1) in high, and as a signal in low.

F can mark nodes as high, low, apex and loose (in the draft and certificate, see below), so that it is visible not only to the node but also to its neighbors (loose, or even apex, status can be omitted, then all apexes, or even all low, would be treated as loose); the algorithm description below uses this recorded high/low status.

Checks. **F** needs to check that its tree has no incident rips, and that the neighbors will not create them after the check is complete. Low nodes —unless loose— need no such checks: they can change neither senior chains nor heights until after the next change to on. Thus, the following checks are performed: In start-1: a loose u rip-checks all its edges before changing to done-1. In active-1: split u rip-check its p_l-pointer (delaying change to done-1). Also a loose active-1 w (which in low occurs before start-1) waits for each low (loose) neighbor to be in phase-0 or active-1 before changing to done-0 (thus assuring correctness of the start-1 check above). In active-0: a high v before changing to done-0 (1) rip-checks all edges, and (2) waits for each (a) high neighbor w to be in phase-0, or to enter active-1 and then enter start-1, (b) low (loose) neighbor w to be in start-0 or active-0 (assuring correctness of the subsequent active-1 check above). Finally, in start-0, loose v waits for the same events as in (2) above before changing to active-0.

Splits: borrowing a pointer. The above checking requires a pointer to "rotate" over the node's neighbors. This (soft) pointer can use the unused hard pointer in the singles or doubles. In splits no spare hard pointer is available, however (instead of adding a hard pointer) we can "borrow" a pointer from the p_b-parent as follows. When a split w needs to use an extra pointer, w requests help from its p_b-parent (low, and thus always single) v. Such v goes around pointing at its needy p_b-children with the "lending" pointer. Such a "lending" pointer on w (there can be at most one), can implement its p_b pointer (in the opposite direction), allowing w to use the corresponding hard pointer for other purposes. When w is done using its client pointer, it can free the "lending" pointer, allowing v to lend it to its other p_b-children. Each split needs to borrow a pointer only when in active-0, so it can request help from its p_b-parent at most once in a **F** cycle, and thus at most two times total before it merges. Since split w might be waiting for its low (loose) neighbors to be in start-0 or active-0, the lending low v should do the lending in the same states (otherwise, a deadlock can occur).

Rip-checking is more efficient if it runs on small groups, called ***clients***. The client tree is formed of the registered p_l when the **F** tree is formed. The subsequent change of the tree to off changes the clients into ***servers***, functioning in a similar fashion (the off may lead to new splits, so the servers are along senior pointer trees). The rip-checking is implemented by interactions of clients and servers as described below. Each client must be large enough to contain its own height (*rise* from the root) ρ; for $\rho = O(1)$ the client is just one node, making its rips instantly detectable. In fact, each client should contain $\theta(\lg \rho)$ nodes and is computed (allocated and initialized) from the parent client.[9] Each client also computes a timer (as in sec. 3.1) which re-checks repeatedly both the client size (compared to its *rise* ρ, which in turn is checked with the parent client) and the upper bound on its computation time (wlog, assume it is $2^t - 1$ for some t; then co-located step counters are trivially assured never to exceed it).

[9] For example, let i be the smallest such that the subtree $T_v(i)$ of all descendants of v at distance $\leq i$ from v contains $|T_v(i)| \geq \lg \rho$ nodes. Then $T_v(i)$ forms a client of v if $|T_v(i)| < 2 \lg \rho$. Otherwise, additional clients are formed (e.g., from the leaves of $T_v(i)$). These additional clients might not be able to form a separate connected subtree, but their nodes can still communicate (as in sec. 3.2) through the nodes of the parent client (thus nodes might need additional child support fields). Finally, subtrees of the nodes which are too small to have clients of their own join the parent, possibly splitting it into more clients similarly to the above. The child support does not introduce any overhead, since similar communication needs to be provided, whether for the own or the child client.

To detect rips, each client is first re-initialized (to assure that it is not created by the adversary) and then goes through its edges one at a time, using a special client pointer, attaching it as a leaf to the server. Each server periodically registers the attached client pointers, then verifies its correctness (from the root), and then serves its height to all the registered clients one bit at a time (the clients that attached to the server after its registration stage are ignored by the server until the next registration). Each client, upon receiving this height, compares it with its own height value. The client-server interface is across the (client pointer) edge connecting them and can work as follows: Let the server height be encoded in ternary, so that no two consequent digits are the same (e.g., we can use "2" as a separator between 0 and 1 digits; more efficiently, to encode the next bit use the two values different from the current one: the greater to encode 1, and the smaller for 0).[10] The step counters and the timer assure that even the adversarially initiated clients and servers terminate promptly ($\Delta(\lg d)^{O(1)}$ after **R** stabilization[11]). If a rip is detected then this and the neighboring trees need to be restructured, so we change the rip servers to *void* to initiate the following restarting procedure, used also in the case of crashing.

5.2 Restarting

A crash might corrupt computations in the clients and servers, so it is safer to reconstruct them, e.g., as follows. Let **F** keep a special reborn flag, typically set to false, but with the default value true. So, when the node is crashed (incl. into a root) and then opened by **R**, it is still reborn. Servers adjacent to a reborn are marked as *void* (starting from the reborn's neighbor and spreading through the whole server tree); cleared server fields in nodes that were crashed (and exited) are also interpreted as *void*. Both high and low start-0 (propagating along the on wave) freezes at the p_l-pointer of a split with a void server, neither crossing the pointer nor changing till the server changes to non-void. If v is adjacent to a void certificate, then v's client-tree (if any) is cleared: v's *void-client* propagates from client-child to its parent until reaching the client's root there the client is cleared, causing the descendant clients to clear as well (the void server's origin also clears its client). If the *void-client* mark (on its way to the root) meets an off wave moving this client to server fields, then the move leaves the resulting server void (since it was just moved from the client fields, this new server does not intersect any clients, so this process does not propagate any further). A reborn flag is cleared when all adjacent servers and clients are void.

When a void server tree has no clients in any of its nodes and no adjacent reborn, **F** computes *re-clients* on the void server tree (similar to clients, but not on p_l-tree). A re-client near a reborn is cleared similarly to the client (the reborn could have been the re-client's child potentially corrupting it): it changes to void re-draft which propagates to the re-client tree root and is erased from there.

When a re-client is constructed, it checks (as part of an echo state propagating from re-client tree leaves to roots, when a node's children are all in echo) that neither reborn nor clients are adjacent; then re-clients are copied to servers (non-void; possibly changing the sign of $h_3 = \pm 0$ at root child accordingly) from the root up the server tree.

5.3 F Performance

In this section all the distances are along the tree edges described in the previous section, and we assume that **R** has stabilized.

A high start-1 changes to active-1 within $O(d)$. Indeed, a high done with start-1 parent changes to start-1 within a step. So, the distance from a high v in start-1 to the nearest done-1 descendant as above (i.e., with no active in between) grows each step till (within $O(d)$) none remain (only start can be a parent of start; similarly, done can have only done children). a high start-1 with neither done nor start-1 children (i.e., only

[10] A client not copying the served bit delays the step in its server parent node (i.e., its mod3 counter is not incremented). Similarly, the server not serving the next bit after the current one is copied delays all its clients' clocks. Thus a client might indirectly delay a different client of the same server. However, since each client has only one server parent, after a server serves a bit, all clients independently and in parallel must consume it promptly, thus avoiding deadlocks. After **R** stabilization, such delays are $O(\Delta \lg d)$; and before it, they do not impact any commitments.

[11] Indeed, if for the client (the same for servers) its $\Delta = O(\lg d)$ then the Δ factor can be ignored; otherwise, if v has $> 2\lg d$ children then these children (without grand-children of v!) form one or more clients of $< 2\lg d$ nodes, whose communication has a Δ delay due to the information going through v, so any polynomial algorithm can be executed by the client in $\Delta(\lg d)^{O(1)}$.

active-1, if any) changes to active-1, so the distance to the furthest high start-1 descendant decreases each step and any high start-1 changes to active-1 within $O(d)$.

a high active-1 changes to done-1 (or off) within $O(d) + \Delta(\lg d)^{O(1)}$. Indeed, each active-1 split must rip-check its p_l (which takes $\Delta(\lg d)^{O(1)}$ steps), after which each high active-1 with all children (if any) in done-1 changes to done-1 within a step, (unless its parent is off).

A loose start-1 v with done-1 children checks the lengths of all its edges within $\Delta^2(\lg d)^{O(1)}$ steps: each edge is rip-checked in $\Delta^2(\lg d)^{O(1)}$ and a client can have $O(\Delta \lg d)$ edges, checked one at a time. Once this rip-check is completed, v changes to done-1.

If a high v in done-1 has a split ancestor with unregistered p_l, then it too changes to done-1 within $O(d) + \Delta(\lg d)^{O(1)}$ and then changes to start-1 or off; and then in $O(d)$ steps more v changes to start-1, or off (and then to start-0) as well. Thus any high v enters active-1 and then start-1 (or changes to start-0). Similarly, a low v in start-1 or done-1 changes to start-0, but with the additional $\Delta^2(\lg d)^{O(1)}$ delay due to the loose nodes.

Let $t_{l1} \stackrel{\text{def}}{=} d + \Delta^2(\lg d)^{O(1)}$ be the time required by a loose w to be seen in phase-0 or active-1. Let $t_{u1} \stackrel{\text{def}}{=} d + \Delta^{(} \lg d)^{O(1)}$ be the time required by an high w to be seen in phase-0, or to enter active-1 and then start-1.

Low active-1, done-0 change to start-0 within $O(\Delta t_{l1})$. Indeed, within $O(d)$ low active-1 has no active-0 descendants: the closest of these changes to active-1 in one step. A loose active-1 changes to done-0 within Δt_{l1}: after waiting for each low (loose) neighbor to be in phase-0 or active-1. A non-loose low active-1 with only done-0 children changes to done-0 in a step, and so the distance to the farthest active-1 decreases. A root with only done-0 children changes to start-1, which changes to start-0, since it has a low descendant, which will change to start-0 too $O(d)$ steps later.

a high start-0 changes to active-0 in $O(d)$. Indeed, any start-0 has no off descendants within $O(d)$. Then a high start-0 with no start-0 children (all, if any, are active-0) changes to active-0, so the distance to the furthest high start-0 descendant decreases each step.

Before a high active-0 can change to done-0 and a loose start-0 to active-0, the rip-checks for the high and neighbor state checks for both high and loose need to be performed. For high, these checks can be done by all the nodes in parallel. Each client needs to check $O(\Delta \lg d)$ edges, each edge checking taking $\Delta(\lg d)^{O(1)}$ steps (plus a delay due to splits borrowing pointers).

In addition to rip-checking, high active-0 and loose start-0 wait $\Delta t_{l1} > t_{u1}$ to see each low (loose) neighbor in start-0 or active-0 (this dominates the check of the high neighbors, which still needs to be performed). Both of these active-0 checks can be done by all high in parallel (with the client restrictions for the rip-check) and both requires pointers (thus splits still need to borrow them from their p_b-parents). The checking of the states dominates the rip-checking, so the time it takes a high active-0 v to check all of its edges is $O(\Delta^2 t_{l1})$. Thus, a split may need to wait for $t_{lend} \stackrel{\text{def}}{=} O(\Delta^3 t_{l1})$ steps before its p_b-parent could lend it the pointer. Thus, all high active-0 v will all complete their checking within $O(\Delta^3 t_{l1})$ and then any high active-0 with no active-0 children will change to done-0. So, within $O(\Delta^3 t_{l1})$ steps ($O(d)$ time for done-0 propagation is absorbed since $d = O(t_{l1})$) all high start-0 change to done-0.

A loose start-0 does not need to borrow a pointer, and so exits to active-0 within $\Delta^2 t_{l1}$. The propagation of active-0, active-1 and done-0 in both directions on the ancestors of loose v takes additional $O(d)$ (absorbed in the asymptotics of t_{l1}). Thus all start-0 change to done-0 within $t_0 \stackrel{\text{def}}{=} O(\Delta^3 t_{l1})$, which also provides the asymptotic upper bound on the **F** cycle time: the time within which **F** turns off at a root (fulfilling (**F**.off)).

5.4 F Correctness

Assuring (**F**.off) is demonstrated above.

Any senior chain contains at most one p_b. Indeed, a split-p_b separates high nodes from low ones, and chains from low nodes can (legally) contain only low (or lock).

A node with off descendants can only be in start-0 or off, together with the above assuring (**F**.cln).

A crash of v marks it reborn, which voids the server trees of v and its neighbors, and clears the client trees adjacent to these void trees. This effectively freezes **F** in the respective nodes. Then reborn it reset to false, and void servers as well as cleared clients are recomputed. Thus, the tree of v and the adjacent trees have new (uncorrupted by crash) servers; the client trees of v and its distance two neighbors are also recomputed and restart their **F** cycles (and will not let **F** turn off when detecting a long edge). Thus, this situation essentially as if the leaves of each of these trees have just changed from off to on (binding corresponding edges), and so it is now reduced to the following.

Assume now no crashes taking place. Consider v changing its senior chain while vw is a rip. Then v is either high or loose: an apex can split, but —unless loose— will go through another **F** cycle before merging (and thus changing its senior chain). Consider the interval from the last moment v was start-0 with an off descendant (there was one that made vw bound) and until **F** turns off before the senior chain change.

F rip-checks all edges incident to high and loose nodes of the tree (start-0 guarantees correctness). Thus, during the rip-check, vw was not long, so w must have changed its height after the rip-check.

If w is high, then v observes it in phase-0, therefore ancestors will rip-check their p_l before **F** turns off at the root (and so before merging). Thus, high w cannot create the rip.

A low w cannot change height unless it is loose. Then v had to wait for w to be in start-0 or active-0. A loose w can change height only if it splits and then merges prematurely: (i) with the new parent u which was on during the split of w, then w merges (possibly without any **F** checks) when changing to off; (ii) with the new parent u which was off$\downarrow\downarrow$ during the split of w, then before w changes to on, some splits pointed at it and u remained non-single, so w merges when changing to on. Before w splits, it rip-checks wu, so if w changes height then u must changes height after the check and before w merges. In case (i) this possibility is eliminated by w waiting (in active-1) for u to be in phase-0 or active-1. Then u rip-checks its p_l-chain if high; if low, u cannot change height either: even if it splits u cannot merge when changing to off (since it has children), and so rip-check of w prevents its change of height. In case (ii) u rip-checks its edges before splitting; if its new parent change height after the check, u would merge prematurely into single, and w would not merge prematurely. Thus w cannot change height.

Therefore, **F** assures (**F**.rip).

Finally, it remains to satisfy (**F**.sgn). This is done by the clients computing $\lambda((h(v)+1)/3)$ in addition to $h(v)$ for each node to be used in case it floats to $\mathsf{h}_3 = 0$.

References

[AKY90] Yehuda Afek, Shay Kutten, Moti Yung. *Memory-efficient self-stabilization on general networks.* In *Workshop on Distributed Algorithms*, 1990.

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

[Bar65] J. M. Barzdin. *The complexity of symmetry recognition by Turing machines.* (in Russian) *Problemi Kibernetiki*, **v. 15**, pp.245-248, 1965.

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

[DGS96] Shlomi Dolev, Mohamed G. Gouda, Marco Schneider. *Memory requirements for silent stabilization.* [PODC], 1996.

[FOCS] *Proc. IEEE Ann. Symp. on the Foundations of Computer Sci.*.

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

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

[Her] Ted Herman. *Self-stabilization bibliography: Access guide.* Chicago J. Theor. Comp. Sci., Working Paper WP-1, initiated Nov., 1996. Also at http://www.cs.uiowa.edu/ftp/selfstab/bibliography/

[IJ90] Amos Israeli, 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.* Colloquia presentations at MIT, IBM, Bellcore, CMU, ICSI Berkeley, Stanford, SRI, UC Davis. 1992. Includes joint results with B. Awerbuch and R. Ostrovsky, and with L. A. Levin (submitted to [FOCS], 1992).

[IL92] Gene Itkis, Leonid A. Levin. *Self-stabilization with constant space.* Manuscript, Nov. 1992 (submitted to [STOC], 1993). Also in [IL94]. Later versions: *Fast and lean self-stabilizing asynchronous protocols.* TR#829, Technion, Israel, July 1994, and in [FOCS], 1994, pp. 226-239.

[IL94] Leonid A. Levin. (Joint work with G. Itkis). *Self-Stabilization.* Sunday's Tutorial Lecture. ICALP, July 1994, Jerusalem.

[Joh97] Colette Johnen. *Memory efficient, self-stabilizing algorithm to construct BFS spanning trees.* [PODC], 1997. Extended version in Proc. *Workshop on Self-Stabilizing System (WSS)*, 1997.

[Le05] Leonid A. Levin. *Aperiodic Tilings: Breaking Translational Symmetry.* Computer J., **48**, 6, 2005.

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

[PODC] *Proc. ACM Ann. Symp. on Principles of Distributed Computing.*

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

[STACS-06] *Proc. 23rd International Symp. on Theor. Aspects of Computer Sci.* Marseille, Feb. 23-25, 2006.

[STOC] *Proc. ACM Ann. Symp. on the Theory of Computation.*

[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, A.Selberg, S.Selberg, K.Thalberg. Universitetsforlaget, 1977.

[Tra64] B. Trakhtenbrot. *Turing computations with logarithmic delay* (in Russian). *Algebra i Logika*, **3**, pp. 33-48, 1964.

APPENDICES

A Sketch for R

$\mathbf{R}$ controls crashed roots (since $\mathbf{R}$ is invoked last, it can crash them back if the roots are uprooted by other protocols) and locks, keeping its own pointers in them. Intuitively, these pointers must always point down, according to the $\mathbf{R}$ own notion of height; the lock ($\mathbf{R}$ pointer) cycles are broken with the help of *acyclicity certificates* (similar to those of [IL92]) maintained in the lock pointer chains. $\mathbf{R}$ crashes its long edges; changing the pointers and requiring adjustment of the certificates. Unlike the clients and servers of $\mathbf{F}$, these certificates must be adjusted locally (on a sufficiently small interval of the certificate: the whole certificate tree is too big). Furthermore, we will define the long edges in such a way that if a configuration has no stubs, it will be guaranteed to have long edges, which can be promptly detected and crashed.

Thus we will reduce $\mathbf{R}$ to (1) C: lock cycle Cutter, and (2) D: Dropper; their performance parameters $\mathsf{t}_{\text{CC}}, \mathsf{t}_{\text{CM}}; \mathsf{t}_{\text{D}}$ are functions of d, Δ, n and sometimes other aspects of the configuration.

A.1 Reduction

Interface. **Fields:** C, D share $p_{\text{C}}, p_{\text{D}}$ in each lock ($\vec{v} \stackrel{\text{def}}{=} v.\vec{p_{\text{D}}} \stackrel{\text{def}}{=} v.p_{\text{D}}$ if $\neq v$, else $v.p_{\text{C}}$; v is a ***root***$_{\mathbf{R}}$ if $\vec{v}{=}v$; d_{D} is the length of the longest $\vec{p_{\text{D}}}$-chain). An additional bit b_l indicates *long* $\vec{p_{\text{D}}}$ (used mainly for the contracts).

Automatic (local) actions: A lock v adjacent to a root$_{\mathbf{R}}{\neq}v$ is crashed if v is root$_{\mathbf{R}}$, or $\vec{v}$ is not a root$_{\mathbf{R}}$, or $v{\neq}v.p_{\text{D}}{\neq}v.p_{\text{C}}{\neq}v$. Crash always loops p_{C}, and sets p_{D} to an adjacent root$_{\mathbf{R}}$ (possibly resulting from an open root) if there is one; if not, p_{D} is looped too (we call such crash ground), except D can also set p_{D} to an adjacent lock with non-loop p_{C}. (So, after the first step, root$_{\mathbf{R}}$ nodes are never adjacent; and for lock $v{=}v.p_{\text{C}}$ either $u{=}v.p_{\text{D}}$ is a root$_{\mathbf{R}}$ or $u.p_{\text{C}}{\neq}u$). A lock v decrements $\mathsf{h}3$ (whenever allowed by the interface of Sec. 4.1) if v is root$_{\mathbf{R}}$ with $v.\mathsf{h}3{\neq}{-1}$, else if $v.\mathsf{h}3{\not\equiv}(\vec{v}).\mathsf{h}3{+}1 \pmod 3$. A lock v sets $v.b_l{\leftarrow}1$ if $(\vec{v}).b_l{=}1$.

Permissions: C is invoked in (and reads fields of) only locks; D acts in all v. C, D can crash any node. D can also set $v.b_l{\leftarrow}1$ of any lock v. When $b_l{=}1$ for v and all its lock $\vec{p_{\text{D}}}$-children, D can change p_{D} to an adjacent lock u with non-looping p_{C} and $u.b_l{=}0$, resetting $v.b_l{\leftarrow}0$. D can loop p_{D}, when $p_{\text{D}}{=}p_{\text{C}}$ and $b_l{=}0$. C can set $p_{\text{C}}{\leftarrow}p_{\text{D}}$ for any lock v. D can also change the sign of $\mathsf{h}3{=}\pm 0$ in locks, and open on locks by swapping $v.p_l, v.p_b$ (both while obeying Interface permissions of Sec. 4.1).

Height. First, let v be a lock. Then $h_{\mathbf{R}}(v) \stackrel{\text{def}}{=} {-1}$ if v is root$_{\mathbf{R}}$, else $h_{\mathbf{R}}(v) \stackrel{\text{def}}{=} h_{\mathbf{R}}(\vec{v}){+}1$ unless $v.b_l{=}1$ — in this case $h_{\mathbf{R}}(v)$ is unchanged from its previous value (undefined before the first action).

Now, let v be open. Then define $h^{(i)}_{\mathbf{R}}(v) \stackrel{\text{def}}{=} h{\in}[-1, 3\cdot 2^i{-}1)$, for unique h such that $w.\mathsf{h}3 \equiv h{+}\rho_{v,w} \pmod 3$ for all w on some (sufficiently long: $O(2^i)$) open p_B-chain from v, where $\rho_{v,w}$ is the chain *rise* from v to w, and if $w.\mathsf{h}3{=}\pm 0$ then its sign is $\lambda((h{+}\rho_{v,w})/3)$, if w is ground then $h{+}\rho_{v,w}{=}w.\mathsf{h}3$. If no $h' \geq 3\cdot 2^i{-}1$ satisfies the same condition on the same chain (intuitively, when the $O(2^i)$ chain contains ground or two marks with non-0 *rise* between them), then we say that $h^{(i)}_{\mathbf{R}}(v)$ is ***final*** and write $h_{\mathbf{R}}(v){=}h$. If $h^{(i)}_{\mathbf{R}}(v)$ is defined but not final, we say $h_{\mathbf{R}}(v) \geq 3\cdot 2^i{-}1$. If more than one $h{\in}[-1, 3\cdot 2^i{-}1)$ satisfies the above condition for the maximal open p_B-chain (the chain is too short, anchored in a lock), then $h^{(i)}_{\mathbf{R}}(v) \stackrel{\text{def}}{=} *$, and $h_{\mathbf{R}}(v)$ is unchanged from its previous value. If not even one such h exists (signs of $\mathsf{h}3{=}\pm 0$ are inconsistent with λ), then $h^{(i)}_{\mathbf{R}}(v) \stackrel{\text{def}}{=} \perp$.

i*-rips.** An edge vu is an ***i*-*rip if (a) v, u are open, $h^{(i)}_{\mathbf{R}}(v){-}h^{(i)}_{\mathbf{R}}(u){\not\equiv}0, \pm 1 \pmod{3{\cdot}2^i}$, or $h^{(i)}_{\mathbf{R}}(v){=}\perp$; or (b) v is a lock with $h_{\mathbf{R}}(v){<}3{\cdot}2^i{-}1$ and $h_{\mathbf{R}}(u){>}h_{\mathbf{R}}(v){+}1$. The i-rip vu is ***fixed*** when u is a lock and $h_{\mathbf{R}}(u){\leq}h_{\mathbf{R}}(v){+}1$. v ***matures*** when ground or root$_{\mathbf{R}}$, when resets $v.b_l{\leftarrow}0$, and after $\mathsf{t}_{\text{D}}(h_{\mathbf{R}}(v))$ steps.

D **commitments:** (1) In mature v, D (a) can reset $v.b_l{\leftarrow}0$ (and change $v.\vec{p_{\text{D}}}$) only if decreasing $h_{\mathbf{R}}(v)$; (b) can open v only with no i-rips, but (c) cannot ground v. (2) D fixes i-rip within $\mathsf{t}_{\text{D}}(2^i)$ ($>\mathsf{t}_{\text{CM}}(2^i)$ below). (3) If orientation remains flat with all non-root$_{\mathbf{R}}$ lock pointers down, then D promptly opens locks.

C **commitments:** (1) After the initial t_{CC} steps, C assures a root$_{\mathbf{R}}$ if there are locks. (2) C un-loops $v.p_{\text{C}}$ in non-root$_{\mathbf{R}}$ lock v within $\mathsf{t}_{\text{CM}}(h_{\mathbf{R}}(v))$. (3) C does not crash $\mathsf{t}_{\text{CM}}(\mathsf{d}_{\text{D}})$. (4) C merges $v.p_{\text{C}} \leftarrow v.\vec{p_{\text{D}}}$ for every lock v within $\mathsf{t}_{\text{CM}}(\mathsf{d}_{\text{D}})$.

A.2 Correctness

Claim A.1 *(D.2) promptly assures stubs.*

This follows directly from the fact that *any configuration with no stubs contains a $\lceil \lg(d{+}1) \rceil$-rip.*

Indeed, set $k{=}\lceil \lg(d{+}1) \rceil$ and let there be no stubs. Then there is p_l-cycle; by (**F**.cln) it is all one phase, thus its **Sh** pointers do not change. By (**LE**.ht), it must also contain a p_l-chain from v to w of *rise* $d{+}1$. If $h^{(k)}_{\mathbf{R}}(w) \not\equiv h^{(k)}_{\mathbf{R}}(v){+}d{+}1 \pmod{3{\cdot}2^k}$, then some p_l in the chain is a k-rip. Else, consider a shortest path $v_0...v_s$, $v_0{=}v$, $v_s{=}w$, $s{\le}d$. Since $s{<}d{+}1{<}3(d{+}1){-}s$, for at least one $j{<}s$ the edge $v_j v_{j+1}$ is a k-rip. ∎

Claim A.2 *(D.2) and (C.1) assure root$_{\mathbf{R}}$ or ground any time after a prompt initial period.*

Indeed, assuming $\mathsf{t}_{\text{CC}}, \mathsf{t}_{\text{D}}({\le}2d)$ are prompt, (D.2) promptly assures a root or root$_{\mathbf{R}}$ if there were no locks initially; otherwise, (C.1) promptly assures root$_{\mathbf{R}}$. A root$_{\mathbf{R}}$ may change only to a root. A root r may uproot; then its p_l-chain leads either to another root, or lock (then root$_{\mathbf{R}}$ is assured by C), or cycle. By (**LE**.ht) the cycle in the last case must be unbalanced, which implies that v was not bound (**F**.rip) and remains ground (since the cycle contains only $\mathsf{b}_{\mathbf{F}}{=}1$ nodes by (**F**.cln)). Furthermore, if there are no more stubs, there must be a $\lceil \lg(d{+}1) \rceil$-rip, which was there even before the uprooting. ∎

For the next claim let us measure time as the number of activations (of any nodes), starting from some initial configuration at time denoted as 0. Let $h_t(v)$ be $h_{\mathbf{R}}(v)$ at time t. We say that node v has (m,h,t)-trajectory if in the 0 to t period (inclusively) the minimum height $h_{\mathbf{R}}(v)$ of v when mature is m, and at the end of this period $h_t(v) = h$.

Claim A.3 *If v has (m,h,t)-trajectory and $h > m+2$ then for any neighbor $w \in \mathbf{E}(v)$ there are $t' < t$, m', h', such that w has (m',h',t')-trajectory and $|m-m'| \le 2, |h-h'| \le 1$.*

Proof: Let v have (m,h,t)-trajectory and $h > m{+}2$. Let t' be the largest such that $h_{t'+1}(v) = h_{t'}(v){+}1 = h$ (i.e., it is the last float to h of the trajectory of v). Then v has $(m,h,t'+1)$-trajectory.

Suppose that the (m',h',t')-trajectory of w violates either $|m-m'| \le 2$ or $|h-h'| \le 1$. Consider the (first) time i when v is at the minimum height $m = h_i(v)$ and floats at the next step $h_{i+1}(v) = m+1$. (Mature v cannot increase $h_{\mathbf{R}}(v)$, other than by floating (D.1); only the first float may be adjacent to rips (**F**.rip).) Since $h > m+1$, v must float again, now to height $m+2$. At that time, $h_{\mathbf{R}}(w)$ will be defined (and $=h(w)$) and will have the value $m+1$ or $m+2$. Thus, $m' \le m+2$. Similar argument provides $m \le m'+2$, showing $|m-m'| \le 2$.

The above implies that at time t' both $h_{t'}(v)$ and $h_{t'}(w)$ are defined. Furthermore, to permit floating of v, we must have $h_{t'}(w)$ be either $h-1$ or h. ∎

Corollary A.4 *If v rises by $d{+}1$ while remaining at $h_{\mathbf{R}}(v){>}2d$ then during that period $h_{\mathbf{R}}(u){>}0$ for all u.*

Proof by induction on distance k from v to (any) u (and using Claim for the inductive step).

Corollary A.5 *If v is a ground or root$_{\mathbf{R}}$, then $h_{\mathbf{R}}(v)$ remains $O(d)$.*

This corollary follows from the previous and Claim A.2 (v is mature after 1 step).

Claim A.6 *Given $v, h_{\mathbf{R}}(v){=}O(d)$, (D.1) promptly assures $h_{\mathbf{R}}(u) = O(d)$ for all u.*

Assume $\mathsf{t}_{\text{D}}(h), \mathsf{t}_{\text{CM}}(h)$ are polynomial in h. Let $v{=}u_0 u_1 \ldots u_k{=}u$ be the shortest path from v to u, and let $h_{\mathbf{R}}(v){\le}h = O(d)$. Then if $h_{\mathbf{R}}(u_i){\le}h{+}i$ then within $O(\mathsf{t}_{\text{CM}}(h{+}i))$ v is open or has a non-loop p_{C} (C.2), and within $O(\mathsf{t}_{\text{D}}(h{+}i))$ more (D.2) assures $h_{\mathbf{R}}(u_{i+1}){\le}h{+}i{+}1$. ∎

Claim A.7 C *and* D *both promptly stop* grounding.

The previous claim implies that all v promptly mature and d_{D} is promptly $O(d)$. Then (D.1c) stops D grounding, and (C.3) promptly stops C grounding. ∎

Claim A.8 *i-rips disappear promptly after* grounding *stops.*

The minimum $h_{\mathbf{R}}(v)$ with i-rip vu increases by (D.2) within $\mathsf{t}_{\text{D}}(2^i)$. ∎

Lemma A.1 D *(and **R**) promptly stabilize.*

After there remains no i-rips for any i (see previous two claims), p_{C} are promptly merged into non-loop p_{D}, so non-root$_{\mathbf{R}}$ locks p_{C} point down. Then, (D.3) assures that locks are opened, stabilizing **R**. ∎

A.3 C sketch

C consists of two protocols *Checker* CC and *Mender* CM, both sharing acyclicity certificate in special lock fields. Intuitively, CC checks certificate crashing p_{C} cycles. CC can also check certificate drafts along $\vec{p}_{\text{D}}$-chains to avoid delayed crashes when the drafts are moved to the official certificates along the (possibly merged) p_{C}-chains. CM mends the certificates when p_{C}-chains change, and extends them to new locks. So, CC write access is limited only to crash. CM reads and writes certificate fields in locks, merges $p_{\text{C}}\leftarrow p_{\text{D}}$ CC promptly (in t_{CC}) breaks any p_{C}-cycle, thus assuring (C.1). CC can verify the correctness of certificate on an k-long chain in poly(k) time, allowing to assure (D.3). CM assures that its modification to the certificates will not harm their correctness (so only ill-initialized certificates and/or processes can cause CC to crash the certificates). When all the certificate chains are short, the certificates can be verified and the CC crashes stop.

CC can use the acyclicity certificates similar to those in [IL92] (see below). Unlike the certificates of **F**, the acyclicity certificates here cannot be reconstructed on the whole tree (as it might be too deep) and so they must be adjusted locally. When one of the endpoints is open, the adjustment is simple: the open node is either crashed into root or the certificate is extended just by one — trivial for many certificates.

A.3.1 Acyclicity Certificates

We illustrate the idea of acyclicity certificates, by briefly sketching a variant used in [IL92]. While there certificate was constructed along the dfs traversal path of a tree, here we define using tree height.

Define $\mu(k) = -0$ iff $\sum_i k_i$ is odd and > 1; $\mu(k) = +0$ otherwise.[12] In section 4 we defined a similar sequence λ. Either of these two (and possibly some others) can be used to break symmetry: We say string $x = x_1x_2\dots x_k$ is *asymmetric* if it has one or two (separated by a special mark) segments of μ or λ embedded in its digits (one sequence bit per constant number of string digits). For simplicity, we ignore other ways to break symmetry. Asymmetry is required for organizing (hierarchical) computations (and for this reason $\lambda(h(v)/3)$ is made available to **R**, D specifically, via $\mathsf{h}3 = \pm 0$).

Let us cut off the tail of each binary string k according to some rule, say, the shortest one starting with 00 (assume binary representation of any k starts with 00). Let us fix a natural representation of all integers $j > 2$ by such tails $\widehat{j}$ and call j the *suffix* $\sigma(k)$ of k. For a string χ, define $\rho(\chi, k)$ to be $\chi_{\sigma(k)}$ if $\sigma(k) \le \|\chi\|$, and special symbol # otherwise. Then $\alpha[k] = \rho(k, k)$, and $\alpha(k) = \langle \alpha[k], \mu(k)\rangle$. [13] Let $\mathcal{L}_\alpha$ be the set of all segments of α. $\mathcal{L}_\alpha$ can be recognized in polynomial time.

Lemma A.2 *Any string of the form ss, $\|s\| > 2$, contains segment $y \notin \mathcal{L}_\alpha$, $\|y\| = (\log\|s\|)^2 + o(1)$.*

Other variants of α can be devised to provide greater efficiency or other desirable properties (e.g., one such variant was proposed in [IL92]).

For a language $\mathcal{L}$ of strings define a $\Gamma(\mathcal{L})$ to be the language of trees, such that any root-leaf path contains a string in $\mathcal{L}$, and any equal length strings on down-paths ending at the same node are identical.

Let $T_A(X_T)$ be a tree T of cellular automata A starting in the initial state with unchanging input X_T. We say that $T_A(X_T)$ *rejects* X_T if some of the automata enter a *reject* state. Language Γ of trees is *t-recognized* by A if for all T, $T_A(X_T)$ (1) rejects within $t(k)$ steps those X_T, which contain a subtree $Y \notin \Gamma$ of depth k; and (2) reject none of the X with all subtrees in Γ. For asynchronous self-stabilizing automata, requirement (1) extends to arbitrary starting configurations and to trees rooted in a cycle; requirement (2) extends to the case when ancestors or children branches of the tree are cut off during the computation.

Lemma A.3 *For any polynomial time language $\mathcal{L}$ of asymmetric strings, $\Gamma(\mathcal{L})$ is recognizable in polynomial time by self-stabilizing protocols on asynchronous cellular tree-automata.*

A.4 D sketch

D maintains groups somewhat similar to servers and clients of **F**. Each group maintains a contiguous segment of an asymmetric sequence (e.g., μ or λ above) and contains the height of (or a lower bound, if

[12] This is a variant of *Thue* (or *Thue-Morse*) sequence [Thu12] defined as $\theta(k) \stackrel{\text{def}}{=} \sum_i k_i \bmod 2$, where k_i is the i-th bit of k.

[13] Inclusion of μ in α makes it asymmetric but otherwise 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. It is also possible to reformulate the definition using λ instead of μ.

near a sufficiently low group). This allows D to hierarchically check for i-rips using the same mechanisms as the acyclicity certificates above. Intuitively, a group, working as a client, checks each of its incident edges one at a time (non-hierarchically, since we are interested only in the groups at $O(d)$ height). However, the servers need to be organized hierarchically, storing also the pointer address in the hierarchical sub-groups to the edges being served. Then even a large group can quickly detect a low adjacent group. For rips with sufficiently large height difference, the subgroup of the appropriate hierarchy level changes the tree as a unit. This may break the original group, but the remaining contiguous segments of asymmetric strings will be sufficiently large to support the subgroups with the sufficiently large lower bounds on height (sufficiently larger than the defecting subgroup's new height).

D extends its the above data structures to the open trees rooted in locks. There, it computes the height using λ embedded in h3$= \pm 0$. If the open tree is not large enough (does not contain two marks with non-0 *rise* between them), nor contains height information written there by D, then D crashes the whole tree. D treats open low and high branches separately: the low subtree is crashed as a group if it has too few nodes to determine the height (even if the high nodes would have added enough nodes).