The Synergy of Finite State Machines (Full Version)

Yehuda Afek*[1], Yuval Emek†[2], and Noa Kolikant[3]

[1]Tel Aviv University. afek@cs.tau.ac.il
[2]Technion. yemek@technion.ac.il
[3]Tel Aviv University. noakolikant@mail.tau.ac.il

Abstract

What can be computed by a network of n randomized finite state machines communicating under the *stone age* model (Emek & Wattenhofer, PODC 2013)? The inherent linear upper bound on the total space of the network implies that its global computational power is not larger than that of a randomized linear space Turing machine, but is this tight? We answer this question affirmatively for bounded degree networks by introducing a stone age algorithm (operating under the most restrictive form of the model) that given a designated *I/O node*, constructs a *tour* in the network that enables the simulation of the Turing machine's tape. To construct the tour with high probability, we first show how to 2-*hop color* the network concurrently with building a spanning tree.

keywords: finite state machines, stone-age model, beeping communication scheme, distributed network computability

*The work of Y. Afek was partially supported by a grant from the Blavatnik Cyber Security Council and the Blavatnik Computer Science Research Fund.

†The work of Y. Emek was supported in part by an Israeli Science Foundation grant number 1016/17.

1 Introduction

Synergy, the whole is greater than its parts, is many times true, however in traditional distributed computing, each node is usually assumed to be as powerful as a Turing machine, hence its local computational power is equivalent to the global computational power of the whole network. Here, we address the computational power of a network of *randomized finite state machines* with a very weak communication scheme (similar to the communication scheme of the *beeping* model, [CK10, AAB+11b]), and show that even under these harsh conditions, synergy can be achieved: *the network as a whole is computationally more powerful than its individual nodes.*

Recently, there is a growing interest in the study of networks of sub-silicon devices, including biological networks [AAB+11a, FK13, NBJ14] and networks of man-made nano-devices [MCS11, DGS+15, CDRR16], through the lens of theoretical distributed computing. These are typically huge networks of primitive devices that nevertheless perform complicated tasks (e.g., an ant colony that solves problems no small number of ants can), thus raising the following question: How do limitations on the local computation and communication capabilities of the individual nodes affect the global computational power of the whole network?

The current paper addresses this question using the *stone age (SA)* model of Emek and Wattenhofer [EW13] that captures a network of devices with very weak local computation and communication capabilities. It has been shown in [EW13, Sec. 5 (full version)] that an n-node SA network with a path topology can simulate a randomized $O(n)$-space Turing machine, denoted hereafter as an RSPACE(n) machine. Little is known though about the global computational power of SA networks with more general topologies and/or more restrictive communication schemes. In this paper, we shed some light into this unexplored research domain, proving that RSPACE(n) machines can be simulated over any network topology of bounded degree by a variant of the SA model where the nodes have no sender collision detection (see Section 1.1).

1.1 Model

Our model follows the *stone age (SA)* model introduced in [EW13] and used subsequently in [ELS+14, AEK18]. Throughout, we assume that the communication network is represented by a finite size connected undirected graph $G = (V, E)$ with node degrees bounded by constant Δ. The nodes are controlled by *randomized finite automata* with state space Q, message alphabet Σ, and transition function τ whose role is explained soon.

Each node $v \in V$ of degree $d_v \leq \Delta$ is associated with d_v *input ports* (or simply *ports*), one port $\psi_v(u)$ for each neighbor u of v in G, holding the last message $\sigma \in \Sigma$ received from u at v. The communication model is defined so that when node u sends a message, the same message is delivered to all its neighbors v; when (a copy of) this message reaches v, it is written into port $\psi_v(u)$, overwriting the previous message in this port. Node v's (read-only) access to its own ports $\psi_v(\cdot)$ is very limited: for each message type $\sigma \in \Sigma$, it can only distinguish between the case where σ is not written in any port $\psi_v(\cdot)$ and the case where it is written in at least one port.

The execution is event driven with an asynchronous scheduler that schedules the aforementioned message delivery events as well as node activation events.[1] When node $v \in V$ is activated, the transition function $\tau : Q \times \{0,1\}^\Sigma \to 2^{Q \times \Sigma}$ determines (in a probabilistic fashion) its next state $q' \in Q$ and the next message $\sigma' \in \Sigma$ to be sent based on its current state $q \in Q$ and the current content of its ports.[2] Formally, the pair (q', σ') is chosen uniformly at random from $\tau(q, \chi_v)$, where $\chi_v \in \{0,1\}^\Sigma$ is defined so that $\chi_v(\sigma) = 1$ if and only if σ is written in at least one port $\psi_v(\cdot)$.

To complete the definition of the randomized finite automata, one has to specify the initial state $q_0 \in Q$ (assumed to be the same for all nodes in the current paper), the set $Q_{out} \subseteq Q$ of output states that encode the node's output, and the initial message $\sigma_0 \in \Sigma$ written in the ports when the execution begins. In addition, SA algorithms are required to have *termination detection*, namely, every node must eventually decide on its output and this decision is irrevocable.

Following the convention in message passing distributed computing (cf. [Pel00]), the *run-time* of an asynchronous SA algorithm is measured in terms of *time units* scaled to the maximum time it takes to deliver any message or the time between any two consecutive activations of a node. Refer to [EW13] for a more detailed description of the SA model.

We adopt the communication scheme presented in [AEK18] that weakens the SA model of [EW13] in two aspects. First, under the model of [EW13], the algorithm designer could choose an additional constant *bounding parameter* $b \in \mathbb{Z}_{>0}$, providing the nodes with the capability to count the number of ports holding message $\sigma \in \Sigma$ up to b. As done in [AEK18], in the current paper, the bounding parameter is set to $b = 1$. This model choice can be viewed as an asynchronous multi-frequency variant of the *beeping* communication model [CK10, AAB+11b].

Second, in contrast to the setting considered in [EW13], the communication graph $G = (V, E)$ assumed in the current paper may include *self-loops* of the form $(v, v) \in E$ which means, in accordance with the aforementioned model definition, that node v admits port $\psi_v(v)$ that holds the last message received from itself. Using the terminology of the beeping model literature (see, e.g., [AAB+11b]), the assumption that the communication graph is free of self-loops corresponds to a *sender collision detection*, whereas lifting this assumption means that node v may not necessarily distinguish its own transmitted message from those of its neighbors $u \neq v$.

The cruxes of the SA model are that (1) node v cannot distinguish between its ports; and (2) the number of states in Q and the size of the message alphabet Σ are constants independent of the size (and any other global parameter) of the graph G.[3] The same holds for the size of the transition function τ, that can be encoded as a $Q \times 2^\Sigma$ table whose entries are subsets of $Q \times \Sigma$.

Sequential Stone Age Machines. We wish to use a SA network to simulate an RSPACE(n) machine $\mathcal{M}$, but before we can describe this simulation, we have to explain how the $O(n)$-bit input

[1] The only assumption we make on the event scheduling is FIFO message delivery: a message sent by node u at time t is written into port $\psi_v(u)$ of its neighbor v before the message sent by u at time $t' > t$.

[2] Node v's actual transition in step t is an atomic operation occurring at the beginning of the step.

[3] In the current paper, $|Q|$ and $|\Sigma|$ may depend on the fixed degree bound Δ; the exact dependency is analyzed in Section 4.3.2.

I of $\mathcal{M}$, that is normally stored in $\mathcal{M}$'s tape at the beginning of the execution, is provided to our network. Clearly, no node in the network can hold more than a constant number of bits, hence I should be distributed over multiple nodes. Unlike [EW13], where a path topology is assumed, here the network topology is arbitrary and does not (initially) induce any sequential order on the nodes, thus storing I in the nodes of the network before the execution starts does not make sense. Instead, we introduce the key notion of a *sequential stone age machine (SSAM)*, where I is fed to the SA algorithm bit-by-bit in a sequential fashion through some node.

Formally, given a network $G = (V, E)$, a SSAM is a SA algorithm running on the nodes of G that allows an external user to
(1) pick any node $v \in V$ and send to it a designated `I/O_prepare` message;
(2) wait until v sends a designated `I/O_ready` message;
(3) feed v with a sequence of input bits by means of sending a sequence of designated input messages (and receiving a corresponding sequence of acknowledgments from v);
(4) wait until the computational process terminates; and
(5) get the desired output back from v by means of receiving from it a sequence of designated output messages.
We refer to node v picked by the user in (1) as an *I/O node*. To exploit the combined computational power of all nodes (in contrast to a single node whose computational power is restricted to that of a randomized finite state machine), the computational process described in (4) typically involves the whole network. The SSAM is said to be a (T^p, T^{io})-*SSAM* if it is guaranteed that the external user waits at most T^p time between sending the `I/O_prepare` message and receiving the `I/O_ready` message and at most T^{io} time between feeding the input bits and getting back the output bits.

1.2 Our Results

We prove that any problem that can be solved whp by an RSPACE(n) machine in time T can be solved whp on any n-node bounded degree graph G by an $(O(D), O(T))$-SSAM operating under an asynchronous scheduler, where D denotes the diameter of G;[4] in other words, it takes $O(D)$ time to initialize the SSAM so that it is ready to accept its input, whereas the actual simulation of the RSPACE(n) machine takes $O(T)$ time. Specifically, our main algorithmic contribution is a SA algorithm that given an n-node bounded degree graph $G = (V, E)$ and a designated *root* node $r \in V$, constructs a 2-*hop coloring* of G and a node sequence $\langle S(i) \rangle_{i=0}^{2n-1}$, referred to as a *tour*, that satisfies: (i) every node appears in S exactly twice; (ii) $S(0) = S(2n-1) = r$; and (iii) the state of node $S(i)$ encodes enough information to route a message to $S(i+1 \mod 2n)$ and to $S(i-1 \mod 2n)$ that reaches its destination in $O(1)$ time for every $0 \leq i \leq 2n-1$; our algorithm terminates with a correct 2-hop coloring and a correct tour in time $O(D)$ whp.

In the SSAM context, the tour S is constructed during phase (2) (while the external user waits for the `I/O_ready` message) with the I/O node serving as the root. This tour is then employed to

[4]Throughout this paper, we say that event A occurs *with high probability*, abbreviated by *whp*, if $\mathbb{P}(A) \geq 1 - n^{-c}$, where n is the number of nodes in the graph and c is an arbitrarily large constant.

simulate a randomized Turing machine $\mathcal{M}$ with a $(2n)$-cell tape in phase (4) as follows. Node $S(i)$ simulates the ith cell $C(i)$ in the tape so that the state of $S(i)$ encodes: (a) the current content of $C(i)$; (b) whether the head of $\mathcal{M}$ is currently located at $C(i)$; and (c) the state of $\mathcal{M}$ in case the head is located at $C(i)$. A step of $\mathcal{M}$'s (sequential) execution, where the head moves from cell $C(i)$ to cell $C(i \pm 1)$, is implemented by sending a designated 'head moving' message from $S(i)$ to $S(i \pm 1)$ that encodes the new state of $\mathcal{M}$.

1.3 Main Technical Challenges

A 2-hop coloring is a useful construction in anonymous networks (see, e.g., [EPSW14]) that enables local point-to-point communication under broadcast communication schemes. As discussed in [EW13, Sec. 4.3], it is fairly easy to design a 2-hop coloring SA algorithm in bounded degree graphs with bounding parameter $b = 2$. However, once the bounding parameter is set to $b = 1$ (as defined in the current paper), this becomes a challenging task because the nodes can no longer verify (deterministically) that their neighborhood does not admit color conflicts. The setting considered in the current paper is even harder since the graph may contain self-loops.

Our algorithm resolves this issue by coloring the nodes concurrently with growing a tree $\widetilde{T}$ of depth $O(D)$ rooted at the designated root r. The nodes use a randomized test that looks for color conflicts and if a conflict is detected, the tree $\widetilde{T}$ is carefully used to reset the coloring and tree construction processes. It is interesting to point out that without a designated root, it is impossible to obtain even a 1-hop coloring in our setting — see Section 5.

Another source of difficulty that we had to overcome when designing our algorithm stems from the requirement that the algorithm terminates correctly whp. While whp guarantees are common in traditional distributed graph algorithms, they are more challenging to obtain with SA algorithms: the individual nodes do not (and cannot) have any notion of n; nevertheless, the algorithm should err with probability that decreases (polynomially) with n.

1.4 Related Work

Most literature on distributed network algorithms does not deal with computability issues, simply because the computational power of the whole network is identical to that of a single node (that is, a Turing machine). Things become more interesting when restrictions are imposed on the computational power of the individual nodes, in particular, when the nodes are restricted to finite state machines.

Computational models based on networks of finite state machines have been studied for many years. The best known such model is the extensively studied *cellular automata* that were introduced by Ulam and von Neumann [vN66] and became popular with Martin Gardner's Scientific American column on Conway's *game of life* [Gar70] (see also [Wol02]). A cellular automaton captures a network of finite state machines, arranged in a grid topology (some other highly regular topologies were also considered), where the transition of each node depends on its current state and the states of its neighbors. The SA model, which is inspired by the cellular automaton model, generalizes

the latter in two aspects: (1) it is applicable to arbitrary network topologies and does not make any regularity assumptions; and (2) it is applicable to fully asynchronous schedulers and does not require that the nodes operate in synchrony (although a small fraction of the cellular automata literature deals with asynchronous node activation [Nak74, AAC+00, Neh03], it does not support the concept of asynchronous message delivery).

Another popular model that considers a network of finite state machines is the *population protocols* model, introduced by Angluin et al. [AAD+06] (see also [AR09, MCS11]), where the network entities communicate through a sequence of atomic pairwise interactions controlled by a fair (adversarial or randomized) scheduler. This model provides an elegant abstraction for networks of mobile devices with proximity derived interactions and it also fits certain types of chemical reaction networks [Dot14].

The neat *amoebot model* introduced by Dolev et al. [DGRS13] also considers a network of finite state machines in a (hexagonal) grid topology, but in contrast to the models discussed so far, the particles in this network are augmented with certain mobility capabilities, inspired by the amoeba contraction-expansion movement mechanism. This model has been successfully employed for the theoretical investigation of self-organizing particle systems [SOP14, DGR+14, DGR+15, DGS+15, DGR+16, CDRR16, DGP+16], especially in the context of *programmable matter*.

The SA model, that constitutes the heart of this paper, was introduced in [EW13] with the goal of demonstrating that certain classic distributed graph problems, e.g., maximal independent set, coloring, and maximal matching, can be solved fast (in polylogarithmic time) by a network of devices whose computation and communication capabilities are perhaps sufficiently weak to capture biological cellular networks (cf. [BPEA+01]). Since then, this model was successfully employed in theoretical studies of distributed computing in other kinds of networks including a series of works inspired by ant foraging processes [LUSW14, ELUW14, LKUW15, ELS+15, CELU17] as modeled by Feinerman et al. [FKLS12, FK12]. The SA maximal independent set algorithm of [EW13] has been adjusted to handle dynamic networks too [EU16].

As discussed in Section 1.2, the SA algorithm developed in the current paper assumes a unique root node, where in the SSAM context, the role of the root node is played by the I/O node chosen by the external user. The task of selecting a unique root node (cf. *leader election*) under the SA model has recently been studied in [AEK18]. This paper also introduces the variant of the SA model (with a weaker communication scheme, see Section 1.1) used in the current paper.

Much of the technical challenge in designing the SA algorithms presented in [EW13, EU16, AEK18] stems from the necessity to handle graphs of arbitrary node degrees. In contrast, the focus of the current paper (that studies a completely different problem) is restricted to bounded degree graphs and the main technical challenge arises from other factors (see Sec. 1.3). In this regard, it is important to point out that although imposing a constant bound on the node degrees is a severe limitation from a graph theoretic perspective, in practice, many of the networks that motivate our model, particularly biological cellular networks, typically exhibit small node degrees.

2 Preliminaries

Notation and Terminology. Throughout this paper, the degree bound Δ is regarded as a constant. In some places, the asymptotic notation is augmented with a Δ subscript to emphasize that a Δ function is hidden, e.g., $O_\Delta(n)$ or $\Omega_\Delta(\log n)$. Using this notation, notice that the diameter D of the graph is larger than $\log_\Delta n = \Omega_\Delta(\log n)$. When the exact dependency on Δ is more important, we may also mention it explicitly inside the asymptotic notation.

We denote the shortest distance (in hops) from node u to node v in graph $G = (V, E)$ by $\text{dist}_G(u, v)$ and omit the subscript G when the graph is clear from the context. The *(inclusive) d-hop neighborhood* of v is denoted by $\Gamma^{d+}(v) = \{u \in V \mid \text{dist}(v, u) \leq d\}$; when $d = 1$, we omit it from the superscript and write simply $\Gamma^+(v)$. Let $\Gamma(v) = \Gamma^+(v) - \{v\}$ be the *(exclusive) neighborhood* of v. Let $\Gamma^*(v) = \Gamma^+(v)$ if v admits a self-loop and $\Gamma^*(v) = \Gamma(v)$ otherwise. A *k-hop coloring* of G is a function $c : V \to \mathbb{Z}_{>0}$ satisfying the requirement that $c(u) \neq c(v)$ for every two nodes $u \neq v$ such that $\text{dist}_G(u, v) \leq k$. A 1-hop coloring is often referred to simply as a *coloring*.

In the context of rooted trees, the terms parent, child, sibling, leaf, depth, and height are used in their standard meaning (see, e.g., [CLRS09]). We also use the term *branch* for a tree path that starts at the root. These terms are generalized from a rooted tree to a *directed acyclic graph (DAG)*, recalling that there, the parent of node v and the branch that ends at v are not necessarily unique. Finally, for a positive integer k, the set $\{1, 2, \ldots, k\}$ is denoted by $[k]$.

Synchronizer Transformation. As explained in Section 1.1, the execution is controlled by an asynchronous scheduler. One of the contributions of [EW13] is a SA *synchronizer* implementation (cf. the α-synchronizer of Awerbuch [Awe85]). Given a synchronous SA algorithm $\mathcal{A}$ whose execution progresses in fully synchronized *rounds* $t \in \mathbb{Z}_{>0}$ (with simultaneous wake-up), the synchronizer generates a valid (asynchronous) SA algorithm $\mathcal{A}'$ whose execution progresses in *pulses* such that the actions taken by $\mathcal{A}'$ in pulse t are identical to those taken by $\mathcal{A}$ in round t.[5] The synchronizer is designed so that the asynchronous algorithm $\mathcal{A}'$ has the same bounding parameter b ($= 1$ in the current paper) and asymptotic run-time as the synchronous algorithm $\mathcal{A}$.

Although the model considered by Emek and Wattenhofer [EW13] assumes that the graph has no self-loops, it is straightforward to apply their synchronizer to graphs that do include self-loops (see also [AEK18]). Hence, our algorithm is designed to operate under a locally synchronous scheduler. Specifically, we assume that the execution progresses in synchronous rounds $t \in \mathbb{Z}_{>0}$, where in round t, each node v

(1) receives the messages sent by its neighbors in round $t - 1$;

(2) updates its state; and

(3) sends a message to its neighbors (same message to all neighbors).

Notice that under a locally synchronous scheduler, if node u is in round t and node v is in round

[5] We emphasize the role of the assumption that when the execution begins, the ports hold the designated initial message σ_0. Based on this assumption, a node can "sense" that some of its neighbors have not been activated yet, hence synchronization can be maintained right from the beginning.

t', then it is guaranteed that $|t - t'| \leq \text{dist}(u, v)$. This means that there exists some constant c so that in any period of $c(T + D)$ time units, every node in the graph performs at least T steps. Consequently, an upper bound of $O(T)$ on the number of rounds performed by an arbitrary node, implies an upper bound of $O(T + D)$ on the total elapsed time.

3 Algorithm Description

In this section, we present our algorithm that constructs the desired $(2n)$-hop long tour over the bounded degree graph assuming there is a single distinguished root node.

Overview. At the beginning, all nodes (other than the root) are identical, and as a first step, the algorithm 2-hop colors them. Given a 2-hop coloring, each node can distinguish between its neighbors and establish a one-to-one communication with each of them. Based on this infrastructure, it is possible to construct a rooted tree from the distinguished root and build a tour on it.

To build the above on a network of identical randomized finite state machines, we use a *layered approach*. The first layer, referred to as the *synchronizer* layer, runs a synchronizer similar to those designed in [AAB+12, EW13, AEK18] so that all layers above it operate assuming a locally synchronous scheduler. The goal of the second layer, referred to as the *degree estimation* layer, is for each node to compute its own degree in G. This is done based on a randomized process that continuously verifies that the current estimate on the number of neighbors is correct. Each time the estimate is updated, a *reset* is invoked and the layers above this second layer may be restarted (more on that later).

In the next, third, 2-*hop coloring* layer, we rely on each node correctly knowing its degree. The root starts a process that 2-hop colors the graph and constructs a rooted spanning tree, denoted by $\widetilde{T}$, at the same time. Upon termination of this process, an echo process up the branches of $\widetilde{T}$ (towards the root) is invoked. Tree $\widetilde{T}$ is designed so that its depth is $O(D)$ whp. Notice that $\widetilde{T}$'s construction assumes all nodes know their accurate degree; if not, the constructed $\widetilde{T}$ may be a DAG (see Section 2), but this will be detected by the degree estimation layer and a reset will (eventually) be invoked to delete $\widetilde{T}$ and restart the tree construction (intertwined with 2-hop coloring) process. The goal of the fourth and final layer, referred to as the *tour construction* layer, is to construct the desired tour. This is done by simulating a *depth first search* traversal of $\widetilde{T}$ in a concurrent manner. The depth first search procedure is implemented so that when the root terminates, the tour is ready to be used.

The layer hierarchy is illustrated in Figure 1;[6] each *block* in this figure represents a layer or a process within a layer. Notice that while the degree estimation and synchronizer blocks work continually, the echo and tour construction blocks are initiated only after the tree construction block terminates.

[6] All figures (and pseudocodes) are deferred to a designated figure appendix.

3.1 The Layers

Recall that initially, all nodes reside in the initial state q_0 and all ports hold the initial letter σ_0. The root starts the synchronizer (as described in Section 2) and any node that reads a message different than σ_0 on any of its ports, starts executing the synchronizer layer and the layers above it as described below. The bottom layer, the synchronizer, is the only asynchronous layer. All layers above it operate under a locally synchronous scheduler and make a transition on a round-by-round basis.

The layers model essentially breaks the state machine in each node into several randomized finite state machines, one per block (a layer or a process within a layer). The state set Q and message alphabet Σ are the Cartesian products of the state sets and message alphabets, respectively, of each block's state machine. Some blocks make no move until another block enters some specified state indicating that its task has been completed (for example, all nodes in the neighborhood of node v must be colored before v can start participating in the tour construction layer).

Below we provide detailed descriptions of the different blocks. For clarity of the exposition, these blocks are described in a conventional (distributed) algorithmic style, but they can be easily implemented using a randomized finite state machine. In Section 4.3.2, we discuss the size of the state sets and message alphabets of those state machines in more detail. Pseudocode descriptions of the different blocks are provided in the designated figure appendix.

3.1.1 The Degree Estimation Layer

In each round (working on top of the synchronizer), each node v first randomly draws a label l_v (the word "color" is preserved for the 2-hop coloring procedure) from a set of Δ^4 labels and then, verifies that the number of distinct labels chosen by its neighbors is not larger than its current `deg_estimate` variable. Otherwise, it updates `deg_estimate` and interrupts the layer above it. These labels are also used in upper layers but are only useful when then number of current labels matches the most updated `deg_estimate`. For this reason we set another flag variable `safe` which is used to indicate if the current number of labels equals `deg_estimate`. See Pseudocode 1 for a pseudocode of the degree estimation layer.

3.1.2 The 2-Hop Coloring Layer

A naive algorithm would be for each node in each round to randomly select a color from a sufficiently large (as a function of Δ) palate of colors until it verifies that the number of colors in its neighborhood is not smaller than the degree estimate given by the layer below. However, neighbors may have to change their color causing a chain of node re-coloring that may be difficult to control. Instead, we carefully use the network's size to our benefit, producing a 2-hop coloring whp. To that end, we *intertwine* the coloring trials with a tree construction process, where each node is colored before joining the tree; this tree, denoted by $\widetilde{T}$, serves as the underlying communication structure if a reset process is invoked (this will be explained later). Using a BFS like tree construction,

we guarantee that the depth of $\widetilde{T}$, as well as its construction time, are $O(D)$ whp. However, the (randomized) 2-hop coloring trials, carried out by the nodes on the boundary of $\widetilde{T}$, may introduce some level of asynchronicity and thus, $\widetilde{T}$ may not be an exact BFS.

The Tree Construction Process. The process of constructing $\widetilde{T}$ while 2-hop coloring the nodes on its boundary is referred to as the *tree construction* process. The main variables maintained by this process for each node v are `color` and `p_color`, storing the color of v and the color of v's parent(s) in $\widetilde{T}$, respectively (the plural form corresponds to the case where $\widetilde{T}$ is a DAG, rather than a tree). The values of these variables are reported in every message sent by v.

The tree construction process starts with the root r setting $r.\texttt{color} \leftarrow 1$; following that, the root transmits a designated `join` message. Node v receiving a `join` message for the first time, initiates its 2-hop coloring trials; when these trials succeed, v joins $\widetilde{T}$.

Specifically, upon receiving the `join` message for the first time, node v selects the color of one of the neighbors $u \in \Gamma(v)$ from which a `join` message was received; it then writes $v.\texttt{p\_color} \leftarrow u.\texttt{color}$ and $v.\texttt{g\_p\_color} \leftarrow u.\texttt{p\_color}$, where `g_p_color` is another (temporary) variable maintained by v. Following that, v randomly selects a new color c from the palette $[\Delta^4 - \{v.\texttt{p\_color}, v.\texttt{g\_p\_color}\}]$ and broadcasts a request to color itself with color c using a designated `color_req`(c) message. Node v then waits (for two rounds) for its neighbor's responses (the process in charge of these responses is described soon), and if receives `approve` messages from all of them, then it writes $v.\texttt{color} \leftarrow c$ and broadcasts a `join` message; otherwise (some neighbors of v did not respond with an `approve` message), it starts a new coloring trial. Once v's `color` variable is set, we think of it as being part of $\widetilde{T}$. Notice that the nodes do not record their children in $\widetilde{T}$. See Pseudocode 2 for a pseudocode of the tree construction process.

The Color Approval Process. The mechanism in charge of approving/disapproving the `color_req` messages is referred to as the *color approval* process. This process runs continually at each node v and its role is to check that when node $u \in \Gamma(v)$ attempts to pick the color c (reflected by a `color_req`(c) message received from u), no other node in $\Gamma^+(v)$ is (or soon to be) colored c. This task is trivial to accomplish if all the nodes in $\Gamma^+(v) - \{u\}$ are already colored. However, it may be the case that multiple neighbors of v may have not yet chosen a color (which means that they have not yet joined $\widetilde{T}$) and thus, may look indistinguishable to v. What if two (or more) of these neighbors attempt to pick the same color c?

To overcome this problem, we go back to the `safe` flag of the degree estimation layer. Recall that this flag is set to true only when the number of distinct labels received by node v is equal to its `deg_estimate` variable, which means that v can distinguish between all its perceived neighbors. Therefore, v sends an `approve` message if (and only if) `safe` == `true` and the colors it receives in the `color_req` messages are consistent with a valid 2-hop coloring given the fixed colors in $\Gamma^+(v)$; that is, the `color_req` colors are distinct and different from the colors of the nodes in $\Gamma^+(v)$ that already fixed their colors. See Pseudocode 3 for a pseudocode of the color approval process.

The Echo Process. An *echo* process up the branches of $\widetilde{T}$ is used to detect the termination of the tree construction process. Each node v can detect the colors of its children in $\widetilde{T}$ — those are simply the nodes declaring v.`color` as their parent's color. When a designated `echo` message is received from all its children (at the same round), v starts sending `echo` messages until its parent sends its own `echo` message. The execution of the echo process together with the tree construction process forces all the nodes of the network to execute at least D rounds, which is critical to guarantee the high probability success of the degree estimation layer as $D > \log_\Delta n$. See Pseudocode 4 for a pseudocode of the echo process.

3.1.3 The Tour Construction Layer

The tour construction layer lays out a $(2n)$-hop long tour of the network based on the $2n$ *timestamps* of a *depth first search (DFS)* traversal of $\widetilde{T}$ (see, e.g., [CLRS09, Section 22.3]). Recall that these timestamps are integers in $\{0, 1, \ldots 2n-1\}$ and that the DFS traversal assigns two of them to each node v: one for the time $0 \leq v.d \leq 2n-2$ at which v is discovered by the DFS traversal and one for the time $0 \leq v.f \leq 2n-1$ at which the DFS traversal backtracks from v. We construct the desired tour $\langle S(i) \rangle_{i=0}^{2n-1}$ so that $S(v.d) = S(v.f) = v$.

The DFS timestamps have the following important property: if one of the timestamps of node u is in $\{v.d \pm 1, v.f \pm 1\}$, then u is either v itself, the parent of v, a child of v, or a sibling of v.[7] The nodes cannot store the actual timestamps (this would have required $\Omega(\log n)$ bits), however, node $S(i)$ can store the colors of the nodes along the unique paths in $\widetilde{T}$ from $S(i)$ to $S(i \pm 1)$. If the 2-hop coloring is valid, then this information enables one-to-one communication between $S(i)$ and $S(i \pm 1)$. The aforementioned property ensures that each one of these two paths includes at most two hops, so in total, the tour construction layer stores at most 8 colors of the palette $[\Delta^4]$ at each node v.

These colors are stored in the designated $\texttt{forward\_pointer}^d$, $\texttt{backward\_pointer}^d$, $\texttt{forward\_pointer}^f$, and $\texttt{backward\_pointer}^f$ variables that correspond to $S(v.d+1)$, $S(v.d-1)$, $S(v.f+1)$, and $S(v.f-1)$, respectively. Each pointer encodes the empty path from v to itself or the path to a child, parent, or sibling of v and thus, consists of at most two colors. Refer to Figure 2 for an illustration of the `forward_pointer` variables in a tree.

The role of the tour construction layer is to set those pointers so that the resulting tour reflects the DFS traversal of $\widetilde{T}$. This is done concurrently at all nodes, without actually running a DFS traversal. This construction could have been totally local, without sending any message, if the nodes would have been fully aware of the color assignment in their 2-hop neighborhood in $\widetilde{T}$. However, this would have been expensive in terms of the size $|Q|$ of the state set (making it exponential, rather than polynomial, in Δ — see Section 4.3.2). Instead, the 2-hop coloring layer constructs $\widetilde{T}$ so that each node stores only its parent. To overcome this obstacle, each node v instructs its children on how to set their pointers, helping them connect to their siblings and to v itself, without knowing all of their colors at the same time. The method employed in this regard is iterative,

[7] The arithmetic done in this section is modulo $2n$.

trading space for time, by connecting one child node at a time. This method is invoked at v right after the echo process has ended (and v stopped sending `echo` messages). See Pseudocode 5 for a pseudocode of the tour construction layer.

3.1.4 Resets

The algorithm described thus far may err if a node in $\widetilde{T}$ under-estimated its degree. In this case the 2-hop coloring may be invalid which means that $\widetilde{T}$ may be a DAG (rather than a tree) and the tour constructed based on $\widetilde{T}$ may be wrong. Therefore, if a degree under-estimation is detected, the data structures of the 2-hop coloring and tour construction layers must be deleted and re-constructed from scratch. This can be detected in two possible ways: (1) when a node in the degree estimation layer increases its `deg_estimate` variable; or (2) when a node receives different messages with the same `color` field (which means that they come from different neighbors that were not distinguished so far). The former condition is checked by the degree estimation layer(see Pseudocode 1); the latter is checked by a designated *validity check* process that runs continually. See Pseudocode 6 for a pseudocode of this process. In both cases, if a degree under-estimation is detected, a designated `reset` interrupt is signaled.

The process that catches the `reset` interrupts is referred to as the *reset* process. As mentioned earlier, this process resets the flags and variables maintained by the 2-hop coloring and tour construction layers. (We emphasize that the `deg_estimate` variable and the variables maintained by the synchronizer layer are not affected by a reset.) But before these flags and variables can be reset at node v, we must make sure that the the reset process is invoked at all other nodes in $\widetilde{T}$. This can be tricky as an independent `reset` interrupt may be signaled at some node u while the reset process initiated by v spreads in the network.

To overcome this obstacle, we follow the reset technique introduced in [AAG87], and execute the reset process as follows. (1) A node receiving a `reset` interrupt initiates a `reset_request` only if it has already joined $\widetilde{T}$ (otherwise no reset is initiated by this node). (2) The `reset_request` messages are forwarded up the branches of $\widetilde{T}$ (towards the root). (3) Upon receiving a `reset_request` message, the root broadcasts `freeze_command` messages that are forwarded to all nodes in $\widetilde{T}$ down its branches. (4) Upon receiving a `freeze_command` message, each leaf echos a `freeze_ack` message; each internal node sends a `freeze_ack` message to its parent after it receives `freeze_ack` messages from all its children. (5) When the root is fully acknowledged with the `freeze_ack` messages (and thus knows the entire $\widetilde{T}$ is frozen), it broadcasts `reset_command` messages that are forwarded to all nodes in $\widetilde{T}$ down its branches. (6) Upon receiving a `reset_command` message, each leaf echos a `reset_ack` message; each internal node sends a `reset_ack` message to its parent after it receives `reset_ack` messages from all its children. At this stage, the (leaf or internal) node resets all flags and variables of the 2-hop coloring and tour construction layers. When the root is fully acknowledged with the `reset_ack` messages, it restarts the 2-hop coloring and tour construction layers from scratch.

Notice that the nodes reset and remove themselves from $\widetilde{T}$ from the leaves up towards the root,

so $\widetilde{T}$ never decomposes into several subgraphs. Moreover, the freeze phase in the reset process ensures that $\widetilde{T}$ does not continue growing in an uncontrolled manner while a reset is in motion. Finally, notice that the reset process works as long as $\widetilde{T}$ is a DAG and does not rely on a tree topology. See Pseudocode 7 for a pseudocode of the reset process.

4 Analysis

Due to space limitations, this extended abstract contains only parts of the analysis; refer to the attached full version for the complete analysis. Recall that the 2-hop coloring layer generates a subgraph $\widetilde{T}$ that consists of all colored nodes with an edge connecting vertex u to vertex v if $u.\texttt{color} = v.\texttt{p\_color}$; for the sake of convenience, we orient this edge from u to v and think of $\widetilde{T}$ as a directed graph.

Lemma 4.1. *$\widetilde{T}$ is a DAG. Moreover, r is the unique source in $\widetilde{T}$.*

Proof. $\widetilde{T}$ is constructed by each node v picking a parent u by saving u's color c. Node u might not be the only neighbor of v colored with c, so when v joins $\widetilde{T}$ an edge between all its neighbors colored with c to v itself is also constructed. There are no cycles because each node v's color is picked to be different from $u.\texttt{color}$ and $u.\texttt{p\_color}$. Each edge in $\widetilde{T}$ is from a parent node toward one of its children. The root does not pick any parent, so it does not have any incoming edges and it is a source. The rest of the nodes in $\widetilde{T}$ pick a parent so they are not sources.

Besides the $\widetilde{T}$ construction process, the reset process is another place in the algorithm where $\widetilde{T}$ is changed and another source vertex might be created. But, the reset process progresses towards the root so the $\widetilde{T}$ never decomposes into several sub graphs, and the root always stays the only source vertex in $\widetilde{T}$. □

4.1 Correctness with High Probability

The desired tour construction relies on the correctness of the 2-hop coloring. The latter allows node v to communicate in a one-to-one fashion with each of its neighbors u, which in turn, supports one-to-one communication with more distant nodes w provided that the colors along some (v, w)-path are known. The correctness (whp) of our 2-hop coloring scheme is based on three foundations:
(1) Once all nodes hold an accurate degree estimation, a valid 2-hop coloring is reached with no further resets with probability 1 (see Corollary 4.4).
(2) After $O(\log_\Delta(n))$ rounds of the degree estimation layer, each node holds an accurate degree estimation whp (see Corollary 4.6).
(3) All nodes execute $\Omega(\log_\Delta(n))$ rounds of the degree estimation layer before termination (see Lemma 4.10).

Lemma 4.2. *Consider a tree construction process that starts after all nodes hold an accurate degree estimation. Let w be some node and let $u, v \in \Gamma^+(w)$, $u \neq v$, be two nodes in $\widetilde{T}$. Then, $u.\texttt{color} \neq v.\texttt{color}$.*

Proof. Nodes u and v are colored since both are in $\widetilde{T}$. Assume by contradiction that $u.\texttt{color} = v.\texttt{color} = c$. Recall that a node can fix its own color only in a round at which it receives `approve` messages from all its neighbors. Let t_u and t_v be these rounds for u and v, respectively, from the perspective of w (that is, in w's round counting). Assume without loss of generality that $t_u \leq t_v$.

If $t_u = t_v$, then w approved $u.\texttt{color} \leftarrow c$ and $v.\texttt{color} \leftarrow c$ in the same round. The design of the color approval process guarantees that $w.\texttt{safe}$ was `true` at that round, which means that the number of distinct labels received by w at that round was $w.\texttt{deg\_estimate}$. The assumption that all `deg_estimate` variables were accurate when the tree construction process started implies that u and v picked distinct labels. But this means that the color approval process at w witnessed a color conflict in that round and did not send an `approve` message, reaching a contradiction.

If $t_u < t_v$, then w approved $v.\texttt{color} \leftarrow c$ after u has already fixed its color to $u.\texttt{color} = c$. But the design of the color approval process guarantees that if v tries to pick color c, then w does not send an `approve` message, again reaching a contradiction. □

Corollary 4.3. *If a tree construction process starts after all nodes hold an accurate degree estimation, then $\widetilde{T}$ will not experience a reset.*

Proof. A reset is initiated either because some node updates its `deg_estimate` variable (the degree estimation layer), which cannot happen due to the assumption, or because two different messages carrying the same `color` field are received (the validity check process), which cannot happen by Lemma 4.2. □

Corollary 4.4. *Once all nodes hold an accurate degree estimation, a valid 2-hop coloring is reached within finite time with probability 1.*

Observation 4.5. *For a label set of size Δ^4, in every round of the degree estimation process, the number of distinct labels received by node v equals its degree with probability $1 - O(1/\Delta^2)$.*

Proof. Follows immediately from a birthday paradox argument since $|\Gamma^*(v)| \leq \Delta + 1$. □

Corollary 4.6. *For a label set of size Δ^4, after $O(\log_\Delta(n))$ rounds of node v's degree estimation layer, v holds an accurate degree estimation whp.*

4.1.1 Enough Rounds

We now turn to establish a lower bound on the number of rounds executed by the degree estimation layer of each node before $\widetilde{T}$'s construction terminates. The correctness of the algorithm will follow by Corollary 4.4 and Corollary 4.6.

The degree estimation layer operates underneath the 2-hop coloring layer and independently of its phase (i.e., tree construction or echo). We shall establish the desired lower bound by analyzing the tree construction and echo processes of the 2-hop coloring layer, showing that they take sufficiently many rounds. To that end, we think of the tree construction and echo processes as a locally synchronous *broadcast-echo* process. The goal in this section is to prove that although this

broadcast-echo process may halt prematurely due to coloring defects, it still takes sufficiently many rounds. In this regard, we ignore (for the time being) the coloring trials that may slow down the broadcast phase of this process even further (recall that we are aiming for a lower bound now).

Lemma 4.7. *Consider some node $v \in V$ at distance $\mathrm{dist}(r, v) = x$ from the root r. Under a locally synchronous scheduler, a broadcast initiated by r at its t_r^0 round arrives to v at its $t_r^0 + x$ round.*

Proof. We mark by $m(x)$ a message that has traveled x hops before it again was sent. For example r' initiative message is $m(0)$ and its children following messages are $m(1)$. We prove the lemma by induction on x.

Message $m(0)$ is sent by r when it initiates the tree construction at its t_r^0 round. Under locally synchronous scheduler, r's neighbors have to proceed to their t_r^0+1 in order to process $m(0)$ because it was sent at someone round t_r^0 of execution. So, $m(1)$ is sent from all nodes distant from r by one hop at their $t_r^0 + 1$ round of execution. Let's assume by induction that all nodes distant from the root by x hops have received and processed $m(x)$ messages at their $t_r^0 + x$ round. Let u be a node such that $\mathrm{dist}(r, u) = x+1$ and let v be a node that sent $m(x)$ to u on v's t_r^0+x round. Node u can process $m(x)$ only when it is at its $t_r^0 + x + 1$ round. When it does, it will respond by broadcasting $m(x + 1)$. □

Color defects in the 2-hop coloring may lead to indistinguishable neighbors in $\widetilde{T}$, thus reducing $\widetilde{T}$ from a rooted tree to a DAG. We have to ensure that this does not result in the "disappearance" of large sub-trees.

Lemma 4.8. *Consider the time at which all nodes have joined $\widetilde{T}$. There exists at least one branch $(r = v_0, v_1, \ldots, v_x)$ in $\widetilde{T}$ with $x = \Omega(\log_\Delta(n))$ such that v_{i-1}, v_i, and v_{i+1} are colored (pairwise) differently for every $0 < i < x$.*

Proof. Let v be a node that maximizes $\mathrm{dist}_G(r, v)$ and let x be its depth in $\widetilde{T}$. By definition, $\mathrm{dist}_G(r, v) \geq \Omega(D) \geq \Omega(\log_\Delta n)$, hence $x \geq \Omega(\log_\Delta n)$. When a node picks a color in the tree construction process, it can pick any color but its parent's color and its grandparent's color, therefore for any $0 < i < x$, it holds that $v_i.\mathrm{color} \neq v_{i-1}.\mathrm{color}$ and $v_i.\mathrm{color} \neq v_{i+1}.\mathrm{color}$. □

Notice that Lemma 4.8 does not imply that v_i can distinguish v_{i+1} from other children it may have in $\widetilde{T}$, but the lemma does ensure that v_i is aware of the fact that it has at least one child whose color is $v_{i+1}.\texttt{color}$. Combined with the properties of the locally synchronous scheduler, we can bound the minimum number of rounds each node completes before the echo process of the 2-hop coloring layer terminates.

Observation 4.9. *Consider the branch $(r = v_0, v_1, \ldots, v_x)$ promised by Lemma 4.8. If v_x completes its role in the echo process at its round t, then node v_i does not complete its role in the echo process before at its round $t + (x - i)$.*

Proof. Follows by a simple induction on $x - i$ since the echo process is designed so that a node transmits its first `echo` message only after all its neighbors transmitted `echo` messages and it stops transmitting `echo` messages once its parent transmits an `echo` message. □

Lemma 4.10. *When the echo process terminates, it is guaranteed that every node performed* $\Omega(\log_\Delta(n))$ *rounds.*

Proof. Suppose that r initiated the tree construction process at its round t. Let v be a node that maximizes $\text{dist}_G(r, v) = H$ and observe that $H = \Omega(D) = \Omega(\log_\Delta(n))$. Let t_1 be v's round at which v received a `join` message for the first time and let t_2 be v's round at which v completed its role in the echo process. We showed in Lemma 4.7 that $t_1 \geq t + H$, so also $t_2 \geq t + H$. By Observation 4.9, we conclude that if r terminates the tree construction process at its t' round, then $t' \geq t_2 + H$, hence $t' - t \geq 2H$.

Since $t > 0$, it follows that the degree estimation layer of r has made at least $2H$ rounds by the time the echo process terminates. Since $\text{dist}_G(r, u) \leq H$ for every node u, the properties of the locally synchronous scheduler ensure that the degree estimation layer of any node has made at least H rounds by that time. The assertion follows as $H = \Omega(\log_\Delta(n))$. □

4.2 Resets

Recall that the DAG $\widetilde{T}$ constructed by the 2-hop coloring layer may be deleted (together with the corresponding 2-hop coloring) due to a reset. In this section, we address the correctness of the reset process, proving that every reset can be mapped injectively to a node that initiated a `reset_request` message and that every node that initiated such a message, performs a reset (and deletes itself from $\widetilde{T}$) within finite time. Moreover, if node v initiates a `reset_request` message over DAG $\widetilde{T}$, then, within finite time, we reach a configuration in which $\widetilde{T}$ is deleted and the network does not contain any reset messages.

Lemma 4.11. *Let* $\widetilde{T}_I$ *be the DAG* $\widetilde{T}$ *when a* `reset_request` *was initiated by one of its nodes and let* $\text{height}(T_I) = H$. *Let* $\widetilde{T}_F$ *be the DAG that* $\widetilde{T}$ *grew into before the reset was completed. Then,* $\text{height}(\widetilde{T}_F) = O(H)$.

Proof. All reset messages proceed one hop at a round with no delays. This rate is at least twice as fast as the rate which nodes join $\widetilde{T}$ because of the inherent delay caused by each node needing to get its color approved. After a node joined $\widetilde{T}$ and picked a new color, it has to wait two executions that assures that the following time it wakes up all responses to its new color are going to be in its ports. This rates difference imply that $\text{height}(\widetilde{T}_F) = O(\text{height}(\widetilde{T}_I)) = O(H)$. □

Lemma 4.11 plays a key role in establishing the following observations as it guarantees that the distance in $\widetilde{T}$ between any two nodes is $O(H)$, irrespective of the stage of the reset process.

Observation 4.12. *If node* v *sends a* `reset_request` *message at round* t, *then it receives a* `freeze_command` *message from some of its parents by round* $t + O(H)$.

Proof. By Lemma 4.1 $\widetilde{T}$ is a DAG with a single source r. Following the logic of Lemma 4.7 we can show that the `reset_request` arrives at r at r's $t' = t + O(H)$ round, and its `freeze_command` response then arrives at v at its $t' + O(H) = t + O(H)$ round. □

Observation 4.13. *If node v sends a* `freeze_command` *message at round t, then it receives* `freeze_ack` *messages from all its children by round $t + O(H)$.*

Proof. Let $(v = v_0, v_1, \ldots, v_x)$ the branch that goes from v to its farthest leaf $v_x \in \widetilde{T}$. By Lemma 4.7, the `freeze_command` arrives at v_x at its $t' = t + O(H)$ round. Following the logic of Observation 4.9, v sends `freeze_ack` at its $t'' = t' + O(H) = t + O(H)$ round. □

Observation 4.14. *If node v sends a* `freeze_ack` *message at round t, then it receives a* `reset_command` *message from some of its parents by round $t + O(H)$.*

Proof. Proof Similar to that of Observation 4.12. □

Observation 4.15. *If node v sends a* `reset_command` *message at round t, then it receives* `reset_ack` *messages from all its children (and performs a reset) by round $t + O(H)$.*

Proof. Proof Similar to that of Observation 4.13. □

Lemma 4.16. *If node v sends a* `reset_request` *message during round t of the root, then there exists a round $t' \leq t + O(H)$ of the root at which $\widetilde{T}$ is deleted completely and the network is clean of reset messages.*

Proof. By Observation 4.12 we know that if v initiated a `reset_request` at its round t, then r initiated a `freeze_command` at some round of r $t_1 = t + O(H)$. Implying Observations 4.14, 4.14 and 4.15 on r we conclude that on some round t' of r it received `reset_ack` from all its children and reset itself, where $t' = t_1 + O(H) = t + O(H)$. At r's t' round $\widetilde{T}$ is deleted completely and no reset message can go over it. □

4.3 Algorithm's Performance and Resources

4.3.1 Run-Time

We divide the algorithm's execution into three stages: stage (1) that lasts from the beginning of the execution until the last reset is over; stage (2) that lasts from the end of stage (1) until the echo process of the 2-hop coloring layer reaches the root; and stage (3) that lasts from the end of stage (2) until the tour construction terminates. Let T_1, T_2, and T_3 be the run-times of the first, second, and third stages, respectively.

Corollary 4.6 guarantees that after each node v executed $O(\log_\Delta(n))$ rounds, it holds an accurate estimation of its degree whp. This means that whp, v will not initiate a new reset after executing $O(\log_\Delta(n))$ rounds. On the other hand, Lemma 4.16 guarantees that a reset initiated by some node will disappear from the network after $O(H)$ rounds of the root, where H is $\widetilde{T}$'s height. In Corollary 4.20, we prove that $H = O_\Delta(D)$, hence $T_1 = O_\Delta(D)$.

The second stage contains the tree construction and echo processes. The run-time of the latter is $O(H)$, which is $O_\Delta(D)$ by Corollary 4.20, whereas the former is intertwined with coloring trials

that may delay its progression and require a more delicate analysis. This analysis is concluded in Corollary 4.19, showing that the run-time of the tree construction process is $O(D+\log n) = O_\Delta(D)$.

The tour construction layer is designed so that once node v completes its role in the echo process of the 2-hop coloring layer, it completes its role in the tour construction layer in $O(\Delta)$ additional rounds. Thus, the run-time of the third stage is $T_3 = O(\Delta)$. To conclude, the total run-time of our algorithm is $T_1 + T_2 + T_3 = O_\Delta(D)$.

A `color_req`$(\cdot)$ trial of node v succeeds when the following two conditions are satisfied for every node $u \in \Gamma^*(v)$: (1) the `safe` flag of u is `true`; (2) u does not witness a `color_req`$(\cdot)$ conflict in $\Gamma^*(u)$. We show that these two conditions are satisfied with probability close to 1.

Lemma 4.17. *Assuming that no `reset` interrupt is signaled in $\Gamma^+(v)$, every `color_req`$(\cdot)$ trial of node v succeeds with probability $1 - O(1/\Delta)$.*

Proof. Fix some node $u \in \Gamma^*(v)$. By Observation 4.5, the `safe` flag of u is `true` with probability $1 - O(1/\Delta^2)$. Since the nodes use a color palette of size Δ^4 and since $|\Gamma^*(u)| \leq \Delta + 1$, we deduce by a birthday paradox argument that the probability that u witnesses a `color_req`$(\cdot)$ conflict in $\Gamma^*(u)$ is $O(1/\Delta^2)$. The assertion follows by a union bound argument over all nodes $u \in \Gamma^*(v)$. □

Lemma 4.18. *Suppose that the root r starts a new tree construction process at round t_0 and let $\widetilde{T}$ be the constructed DAG. If node v with $\mathrm{dist}_G(r, v) = x$ joins $\widetilde{T}$ during round t_1 of r, then whp (1) $t_1 - t_0 \leq O(x + \log n)$; and (2) the depth of v in $\widetilde{T}$ is $O(x + \log n)$.*

Proof. Let $\pi = (r = v_0, v_1, \ldots, v_x = v)$ be a shortest (r, v)-path in G. When a node is trying to pick a color, it may go through successive `color_req`$(\cdot)$ trials until one of them is approved. The actual order in which the nodes in π are colored is not necessarily identical to the order induced by π, but after x `color_req`$(\cdot)$ trails in π are approved, all nodes in π are colored.

Assuming that no `reset` interrupt is signaled in $\Gamma^+(v)$, we prove in the attached full version that every `color_req`$(\cdot)$ trial of node succeeds with probability $1 - O(1/\Delta)$. The number of failed `color_req`$(\cdot)$ trials in π is stochastically dominated by a negative binomial random variable N with parameters x and $1 - p$, where $p = 1 - O(1/\Delta) \geq \Omega(1)$ Therefore, we have to up-bound $N + x$ which accounts for the total number of `color_req`$(\cdot)$ trails in π, including the approved ones. Using standard tail bounds on the negative binomial distribution, we conclude that $N + x = O(x + \log n)$ whp.

Starting from r's round t_0, after r executed at least $(c+1)x + c \cdot \log(n) = O(x + \log n)$ rounds, it is ensured by the properties of the locally synchronous scheduler that all nodes at distance at most x from r executed at least $c(x + \log n)$ rounds. Therefore, all these nodes have fixed their color and joined $\widetilde{T}$ whp, so their depth in $\widetilde{T}$ is at most $O(x + \log n)$. □

Corollary 4.19. *Suppose that the root r starts a new tree construction process at round t_0 after all nodes hold an accurate degree estimation. Then, this process is completed within $O(D + \log n)$ rounds of r.*

Corollary 4.20. *Any DAG $\widetilde{T}$ constructed by the tree construction process satisfies* $\text{height}(\widetilde{T}) \leq O_\Delta(D)$ *whp.*

Proof. Let v be a node at distance $x \leq D$ from r. By Lemma 4.18, we conclude that v joins $\widetilde{T}$ in r's $t_0 + O(D + \log n)$ round whp, implying that its depth in $\widetilde{T}$ is $O(D + \log n) = O_\Delta(D)$. □

4.3.2 Protocol's Constants

When discussing the resources of a SA algorithm, we distinguish between the state space size $|Q|$ and the communication alphabet size $|\Sigma|$. In the development of the algorithm, we strove to reduce $|Q|$ and $|\Sigma|$ to make them polynomial in Δ which means that each state in Q and each message in Σ can be encoded with $O(\log \Delta)$ bits. In this section we go through the algorithm's layers and explain in general terms why $|Q|$ and $|\Sigma|$ are indeed polynomial in Δ.

Recall that the state space Q and the alphabet Σ are the Cartesian products of the individual layers' state spaces and alphabets, respectively. We establish the desired upper bounds by showing that each layer requires (i) a state space of size $\Delta^{O(1)}$; and (ii) $\Delta^{O(1)}$ different message types. (We emphasize that these $O(1)$ expressions are universal constants that do not hide a dependency on Δ.)

Observation 4.21. *The degree estimation layer requires a state space of size $O(\Delta)$ and $O(\Delta^3)$ different message types.*

Proof. First, this layer's state holds its current degree estimation and if it has a new estimation from last round. This sums up to $2 \cdot \Delta$ states. Second, the labels that are encoded in the single message of the layers are taken from the labels' set is size Δ^3. □

Observation 4.22. *The 2-hop coloring layer requires a state space of size $O(\Delta^8)$ and $O(\Delta^{12})$ different message types.*

Proof. This layer's state embody the following information: (1) node's phase in the layer's algorithm (waiting for a `join` message, trying to color itself, waiting for other nodes responses, waiting for subtree acknowledgment, and different reset phases). These all sum up to a constant c_1. (2) node's data: color, parent's color and if it is a root node. Totally the needed states amount is $O(\Delta^{2 \cdot 4})$.

Regrading the alphabet, each message sent by a node contains: its color (as a source address), receivers color (destination address) its parent's color, its 2-hop-neighborhood coloring status (accept/ decline/ cannot decide.) and message type (`join` , `color_req`, `tree_creation_ack`, reset messages). So, the message contains three colors and other fields that are taken from small sets of options. So, this layer requires $O(\Delta^{3 \cdot 4})$ messages. □

Observation 4.23. *The tour construction layer requires a state space of size $O(\Delta^{16})$ and $O(\Delta^{12})$ different message types.*

Proof. The tour construction layer's state holds the tour representation by four pointers, each is a color, so together this layer requires $O(\Delta^{4 \cdot 4})$ states. The messages are instructions on how to fill

pointers containing the source color, the destination color, the instruction type and another color to fill a pointer with. Together this layer's possible messages amount is $O(\Delta^{3\cdot 4})$. □

5 The Necessity of our Assumptions

As discussed in Sec. 1.1, from the perspective of the algorithm designer, the model considered in the current paper is weaker than that of [EW13] in the sense that the bounding parameter is restricted to $b = 1$ and the graph may contain self loops (cf. [AEK18]). At the same time, the current paper makes two simplifying assumptions:
(1) the node degrees are bounded by some constant Δ; and
(2) the algorithm is provided with a unique I/O node and although this node is chosen arbitrarily, it may still serve as a (unique) *leader*, thus facilitating the design of the SSAM.
Assumption (1) is clearly mandatory for the construction of the 2-hop coloring as the number of colors is an inherent lower bound on the number of states. Whether a 2-hop coloring is indeed a prerequisite for a SSAM is left as an open question; we conjecture that the answer to this question is positive. Assumption (2) is justified by the following two lemmas.

Lemma 5.1. *At the absence of a designated root node, there does not exist any SA algorithm that solves the (1-hop) coloring problem on a simple path with self-loops with failure probability bounded away from* 1*.*

Lemma 5.2. *At the absence of a designated root node, there does not exist any SA algorithm that solves the 2-hop coloring problem on a simple path without self-loops with failure probability bounded away from* 1*.*

The proofs of Lem. 5.1 and 5.2 are based on probabilistic indistinguishability arguments, similar to those used in many distributed computing negative results, starting with the classic leader election impossibility result of Itai and Rodeh [IR90] (see also [AEK18]).

Proof of Lem. 5.1. Our attention in this proof is restricted to algorithms operating under a fully synchronous scheduler on graph family $\{L_n^{\circlearrowleft}\}_{n\geq 1}$, where $L_n^{\circlearrowleft}$ is a simple path of n nodes augmented with self-loops. Assume by contradiction that there exists an algorithm $\mathcal{A}$ as in the lemma's statement and let Σ denote its message alphabet. Consider the execution of $\mathcal{A}$ on the instance $L_1^{\circlearrowleft}$ and let v be the (single) node in this instance. By definition, there exist a color c, constants $p > 0$ and ℓ, and message sequence $S \in \Sigma^{\ell}$ such that when $\mathcal{A}$ runs on this instance, with probability at least p, node v reads message $S(t)$ in its (single) port in round $t = 1, \ldots, \ell$ and chooses color c at the end of round ℓ.

Now, consider graph $L_n^{\circlearrowleft}$ for some sufficiently large n whose value is determined later on and let $v_1, \ldots, v_{2\ell+2}$ be any $2\ell + 2$ contiguous nodes in the underlying path of $L_n^{\circlearrowleft}$, referred to as a *gadget*. The key observation now is that when $\mathcal{A}$ runs on $L_n^{\circlearrowleft}$, with probability at least $q = p^{2\ell+2}$, both middle nodes $v_{\ell+1}$ and $v_{\ell+2}$ in the gadget receive the same message $S(t)$ in (all) their ports in round

$t = 1, \ldots, \ell$ and both choose color c at the end of round ℓ, independently of the random bits of the nodes outside the gadget. We refer to this event, which clearly leads to an invalid output, as a gadget *failure*. Since p and ℓ are constants that depend only on $\mathcal{A}$, $q = p^{2\ell+2}$ is also a constant that depends only on $\mathcal{A}$.

Take z to be an arbitrarily large constant. If n is sufficiently large, then we can embed $y = \lceil z/q \rceil$ disjoint gadgets in $L_n^{\circlearrowleft}$. When $\mathcal{A}$ runs on $L_n^{\circlearrowleft}$, each of these y gadgets fails (independently) with probability at least q. Therefore, the probability that $\mathcal{A}$ returns a valid output is at most $(1-q)^y$. The assertion follows since this expression tends to 0 as $y \to \infty$ which is obtained as $z \to \infty$. □

The proof of Lem. 5.1 essentially shows that no SA algorithm can distinguish between $L_1^{\circlearrowleft}$ and $L_n^{\circlearrowleft}$ with a bounded failure probability. Regarding Lem. 5.2, we can use a very similar line of arguments to show that no SA algorithm can distinguish between L_2 and L_n with a bounded failure probability, thus establishing the lemma.

References

[AAB+11a] Y. Afek, N. Alon, O. Barad, E. Hornstein, N. Barkai, and Z. Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.

[AAB+11b] Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. In *DISC*, pages 32–50, 2011.

[AAB+12] Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *CoRR*, abs/1206.0150, 2012.

[AAC+00] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T.F. Knight, Jr., R. Nagpal, E. Rauch, G.J. Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.

[AAD+06] D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, 2006.

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 358–370, Washington, DC, USA, 1987. IEEE Computer Society.

[AEK18] Yehuda Afek, Yuval Emek, and Noa Kolikant. Selecting a leader in a network of finite state machines. In *DISC*, 2018. The full version can be obtained from http://arxiv.org/abs/1805.05660.

[AR09] J. Aspnes and E. Ruppert. An introduction to population protocols. In Benoît Garbinato, Hugo Miranda, and Luís Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer-Verlag, 2009.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[BPEA+01] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430–434, 2001.

[CDRR16] S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A markov chain algorithm for compression in self-organizing particle systems. In *PODC*, pages 279–288, 2016.

[CELU17] Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an infinite space with finite memory scouts. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 207–224, 2017.

[CK10] Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *DISC*, pages 148–162, 2010.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[DGP+16] Z. Derakhshandeh, R. Gmyr, A. Porter, A.W. Richa, C. Scheideler, and T. Strothmann. On the runtime of universal coating for programmable matter. In *DNA*, pages 148–164, 2016.

[DGR+14] Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, T. Strothmann, and S. Tzur-David. Infinite object coating in the amoebot model. *CoRR*, abs/1411.2356, 2014.

[DGR+15] Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *NANOCOM*, pages 21:1–21:2, 2015.

[DGR+16] Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal shape formation for programmable matter. In *SPAA*, pages 289–299, 2016.

[DGRS13] S. Dolev, R. Gmyr, A.W. Richa, and C. Scheideler. Ameba-inspired self-organizing particle systems. *CoRR*, abs/1307.4259, 2013.

[DGS+15] Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A.W. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA*, pages 117–132, 2015.

[Dot14] D. Doty. Timing in chemical reaction networks. In *SODA*, pages 772–784, 2014.

[ELS+14] Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How Many Ants Does It Take To Find the Food? In *21th International Colloquium on Structural Information and Communication Complexity (SIROCCO), Hida Takayama, Japan*, July 2014.

[ELS+15] Y. Emek, T. Langner, D. Stolz, J. Uitto, and R. Wattenhofer. How many ants does it take to find the food? *Theor. Comput. Sci.*, 608:255–267, 2015.

[ELUW14] Y. Emek, T. Langner, J. Uitto, and R. Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *ICALP*, pages 471–482, 2014.

[EPSW14] Yuval Emek, Christoph Pfister, Jochen Seidel, and Roger Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 96–105, 2014.

[EU16] Y. Emek and J. Uitto. Dynamic networks of finite state machines. In *SIROCCO*, pages 19–34, 2016.

[EW13] Y. Emek and R. Wattenhofer. Stone age distributed computing. In *PODC*, pages 137–146, 2013. The full version can be obtained from http://yemek.net.technion.ac.il/files/stone-age.pdf.

[FK12] O. Feinerman and A. Korman. Memory lower bounds for randomized collaborative search and implications for biology. In *DISC*, pages 61–75, 2012.

[FK13] O. Feinerman and A. Korman. *Theoretical distributed computing meets biology: a review*, pages 1–18. Springer Berlin Heidelberg, 2013.

[FKLS12] O. Feinerman, A. Korman, Z. Lotker, and J.S. Sereni. Collaborative search on the plane without communication. In *PODC*, pages 77–86, 2012.

[Gar70] M. Gardner. The fantastic combinations of john conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123, 1970.

[IR90] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.

[LKUW15] T. Langner, B. Keller, J. Uitto, and R. Wattenhofer. Overcoming obstacles with ants. In *OPODIS*, pages 9:1–9:17, 2015.

[LUSW14] T. Langner, J. Uitto, D. Stolz, and R. Wattenhofer. Fault-tolerant ANTS. In *DISC*, pages 31–45, 2014.

[MCS11] O. Michail, I. Chatzigiannakis, and P.G. Spirakis. *New models for population protocols*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

[Nak74] K. Nakamura. Asynchronous cellular automata and their computational ability. *Syst Comput Controls*, 5(5):58–66, 1974.

[NBJ14] S. Navlakha and Z. Bar-Joseph. Distributed information processing in biological and computational systems. *Commun. ACM*, 58(1):94–102, 2014.

[Neh03] C.L. Nehaniv. Asynchronous automata networks can emulate any synchronous automata network. *Journal of Algebra*, pages 1–21, 2003.

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[SOP14] NSF workshop on self-organizing particle systems (SOPS). http://sops2014.cs.upb.de/, 2014.

[vN66] J. von Neumann. *Theory of self-reproducing automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[Wol02] S. Wolfram. *A new kind of science*. Wolfram Media, Champaign, Illinois, 2002.

FIGURES

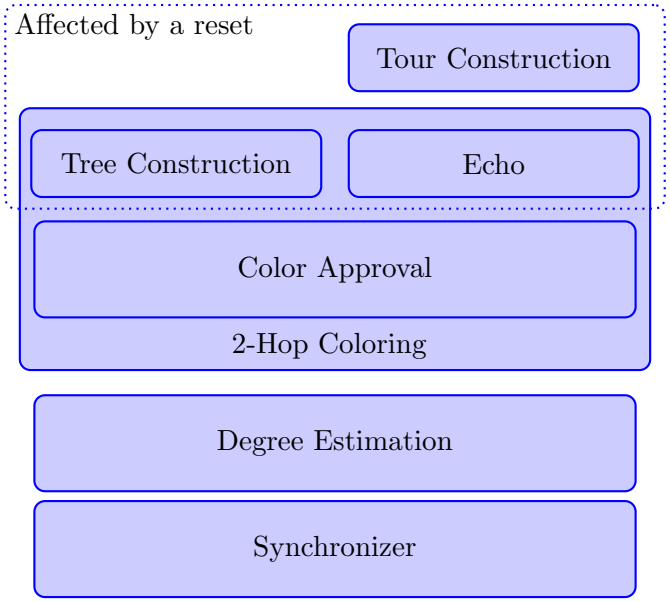

Figure 1: The layers Hierarchy. The blocks correspond to the layers and the processes within the layers. The horizontal axis represents sequential dependencies in the algorithm, whereas the vertical axis represents parallel execution.

Pseudocode 1 The operation of the degree estimation layer at node v.

1: pick a random label $\ell_v \in_r [\Delta^4]$
2: transmit ℓ_v
3: receive ℓ_u from each $u \in \Gamma^*(v)$
4: $\lambda \leftarrow$ number of distinct ℓ_u messages
5: **if** $\lambda > v.$`deg_estimate` **then**
6: update $v.$`deg_estimate` $\leftarrow \lambda$
7: signal the 2-hop coloring and tour construction layers with a `reset` interrupt
8: **else if** $\lambda == v.$`deg_estimate` **then**
9: $v.$`safe` $\leftarrow$ `true`
10: **else** $v.$`safe` $\leftarrow$ `false`

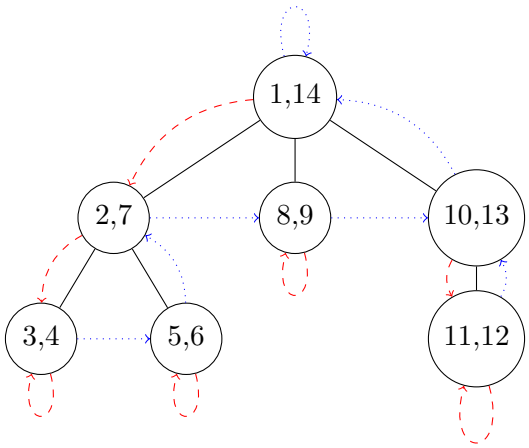

Figure 2: Each node depicts its $v.d, v.f$ timestamps. The dashed red (resp., dotted blue) arrows correspond to the $\texttt{forward\_pointer}^d$ (resp., $\texttt{forward\_pointer}^f$) variables.

Pseudocode 2 The operation of the tree construction process at node v with $v.\texttt{color} == \texttt{null}$.

1: **if** received `join` message m and $v.\texttt{join\_received} == \texttt{false}$ **then**
2: $v.\texttt{p\_color} \leftarrow \texttt{m.color}$
3: $v.\texttt{g\_p\_color} \leftarrow \texttt{m.p\_color}$
4: $v.\texttt{join\_received} \leftarrow \texttt{true}$
5: **if** $v.\texttt{join\_received} == \texttt{true}$ **then**
6: pick a random color $c \in_r [\Delta^4] - \{v.\texttt{p\_color}, v.\texttt{g\_p\_color}\}$
7: transmit $\texttt{color\_req}(c)$
8: wait for responses
9: **if** all received messages are `approve` **then**
10: $v.\texttt{color} \leftarrow c$
11: transmit `join`

Pseudocode 3 The operation of the color approval process at node v.

1: **if** $v.\texttt{safe} == \texttt{true}$ **then**
2: $M \leftarrow$ distinct received messages
3: **for all** $\texttt{color\_req}(c)$ message $m \in M$ **do**
4: **if** exists $\texttt{color\_req}(c)$ message $m' \in M$, $m' \neq m$ **then** return ▷ disapprove
5: **if** exists message $m' \in M$ with $m'.\texttt{color} == c$ **then** return ▷ disapprove
6: transmit `approve`

Pseudocode 4 The operation of the echo process at node v.

if v.`active_echo` == `false` **then**
    if all received messages m with m.`p_color` == v.`color` are `echo` messages **then**
        v.`active_echo` $\leftarrow$ `true`
if v.`active_echo` == `true` **then**
    transmit `echo`
    $m^p \leftarrow$ received message with m^p.`color` == v.`p_color`
    if m^p is an `echo` message **then**
        v.`active_echo` $\leftarrow$ `false`

Pseudocode 5 The operation of the tour construction layer at node v.

$m^p \leftarrow$ received message with m^p.`color` == v.`p_color`
$M \leftarrow$ received messages m with m.`p_color` == v.`color`
let $M = \{m_1, \ldots, m_k\}$ so that m_i.`color` $< m_{i+1}$.`color` for $i = 1, \ldots, k-1$
if $|M| == 0$ **then**
    v.`forward_pointer`$^d \leftarrow \langle\rangle$
    v.`backward_pointer`$^f \leftarrow \langle\rangle$
else
    v.`forward_pointer`$^d \leftarrow \langle m_1$.`color`$\rangle$
    v.`backward_pointer`$^f \leftarrow \langle m_k$.`color`$\rangle$
if m^p is an `instruct`(c, c') **then**
    if $c == v$.`color` **then**
        v.`forward_pointer`$^f \leftarrow \langle v$.`p_color`$, c'\rangle$
    if $c' == v$.`color` **then**
        v.`backward_pointer`$^d \leftarrow \langle v$.`p_color`$, c\rangle$
transmit `instruct`(v.`color`, m_1.`color`)
transmit `instruct`(m_k.`color`, v.`color`)
for $i = 1, \ldots, k-1$ **do** ▷ one iteration per round
    transmit `instruct`(m_i.`color`, m_{i+1}.`color`)

Pseudocode 6 The operation of the validity check process at node v.

for all received messages m **do**
    for all received messages $m' \neq m$ **do**
        if m.`color` == m'.`color` **then**
            signal the 2-hop coloring and tour construction layers with a `reset` interrupt

Pseudocode 7 The operation of the reset process at node v.

```
1: if reset interrupt received then                                   ▷ interrupt handler
2:     if v.color ≠ null then                                         ▷ v is already in T̃
3:         transmit reset_request
4:         v.state ← need_freeze_command
5: if received reset_request message m with m.p_color == v.color ≠ null then
6:     transmit reset_request
7:     v.state ← need_freeze_command
8: if v.state == need_freeze_command then
9:     if received freeze_command message m with m.color == v.p_color then
10:        v.state ← need_freeze_ack
11:        transmit freeze_command
12: if v.state == need_freeze_ack then
13:    if all received messages m with m.p_color == v.color are freeze_ack messages then
14:        v.state ← need_reset_command
15:        transmit freeze_ack
16: if v.state == need_reset_command then
17:    if received reset_command message m with m.color == v.p_color then
18:        v.state ← need_reset_ack
19:        transmit reset_command
20: if v.state == need_reset_ack then
21:    if all received messages m with m.p_color == v.color are reset_ack messages then
22:        v.state ← need_halt
23:        transmit reset_ack
24: if v.state == need_halt then
25:    transmit reset_ack
26:    if received reset_ack message m with m.color == v.p_color then
27:        reset flags and variables of the 2-hop coloring and tour construction layers
28:        halt reset process at v
```