Theoretical Computer Science 410 (2009) 514–532

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Empire of colonies: Self-stabilizing and self-organizing distributed algorithm☆

Shlomi Dolev, Nir Tzachar *

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel

A R T I C L E I N F O

Keywords:
Self-stabilizing
Self-organizing
Communication and routing
Clustering

A B S T R A C T

Self-stabilization ensures automatic recovery from an arbitrary state; we define *self-organization* as a property of algorithms which display local attributes. More precisely, we say that an algorithm is self-organizing if (1) it converges in sublinear time and (2) reacts "fast" to topology changes. If $s(n)$ is an upper bound on the convergence time and $d(n)$ is an upper bound on the convergence time following a topology change, then $s(n) \in o(n)$ and $d(n) \in o(s(n))$. The self-organization property can then be used for gaining, in sub-linear time, global properties and reaction to changes. We present self-stabilizing and self-organizing algorithms for many distributed algorithms, including distributed snapshot and leader election.

We present a new randomized self-stabilizing distributed algorithm for cluster definition in communication graphs of bounded degree processors. These graphs reflect sensor networks deployment. The algorithm converges in $O(\log n)$ expected number of rounds, handles dynamic changes locally and is, therefore, *self-organizing*. Applying the clustering algorithm to specific classes of communication graphs, in $O(\log n)$ levels, using an overlay network abstraction, results in a self-stabilizing and self-organizing distributed algorithm for hierarchy definition.

Given the obtained hierarchy definition, we present an algorithm for hierarchical distributed snapshots. The algorithms are based on a new basic snap-stabilizing snapshot algorithm, designed for message passing systems in which a distributed spanning tree is defined and in which processors communicate using bounded links capacity. The algorithm is *on-demand* self-stabilizing when no such distributed spanning tree is defined. Namely, it stabilizes regardless of the number of snapshot invocations.

The combination of the self-stabilizing and self-organizing distributed hierarchy construction and the snapshot algorithm forms an efficient self-stabilizer transformer. Given a distributed algorithm for a specific task, we are able to convert the algorithm into a self-stabilizing algorithm for the same task with an expected convergence time of $O(\log^2 n)$ rounds.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

The availability and robustness, as well as the possibility for on-demand reconfiguration of large systems, are in many cases vital; be it clusters of servers that support commercial activity, a grid of computers that participate in a complicated computation or a dynamic sensor network. In particular, an important aspect for large on-going systems is the ability to

☆ An extended abstract of this worked appeared in OPODIS 2006.

* Corresponding author.
E-mail addresses: dolev@cs.bgu.ac.il (S. Dolev), tzachar@cs.bgu.ac.il (N. Tzachar).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.10.006

automatically recover from an inconsistent state, namely to be *self-stabilizing* [8,9] or in other words, to have a system that can be started in an arbitrary state.

To capture the need of the industry in autonomic and self-* systems, we propose combining self-stabilization (in fact SuperStabilization [10]) with *self-organization*. While self-stabilization is well defined, the self-organization property has no widely agreed upon definition. We propose to define self-organization as satisfying two main properties: locality and dynamicity. Namely, we require that (1) the algorithm stabilizes in sublinear time with regard to the number of processors and that (2) the addition and removal of processors influences a small number of other processors' states. In other words, if $s(n)$ represents the stabilization time and $d(n)$ represents an upper bound on the stabilization time (and number of state changes) following a dynamic topology change, then: $s(n) \in o(n)$ and $d(n) \in o(s(n))$. This definition can be naturally extended to also capture the effect of local transient faults that corrupt the states of a subset of the processors rather than only topology changes (thus it is in the spirit of both the superstabilizing and fault-containment approaches [9]).

In this work, we allow algorithms to define (on the fly) and (then immediately) use *hyper communication links*, which are overlay links that are constructed of communication links along a path. We regard the time that a message travels over such a link as one time unit, assuming that (practically) no processing is involved in forwarding messages over these links (e.g., [12,26], MPLS [5]). This definition is motivated by (e.g., telephony) systems, where switches along a path are configured for a session and the path is essentially a wire. We propose to use the self-stabilizing and self-organizing properties of our schemes combined with such switching capabilities to obtain dramatically faster convergence rates and global information transmission with relation to traditional communication networks. In traditional settings, there are obvious lower bounds that are proportional to the number of nodes (or the diameter of the communication graph of the system), while the existence of overlay links allows us to obtain logarithmic bounds.

1.1. Main contribution

Self-stabilizing and self-organizing hierarchy definition. The hierarchy of subsystems is defined by partitioning the communication graph into small clusters, after which clusters are merged to form larger clusters and so on. The partition can be done according to a designer's input, using an automatic off-line clustering algorithm or even an on-line clustering algorithm that reflects the system's current behavior. In particular, we suggest a randomized self-stabilizing and self-organizing partition that is based on periodical collection of local topology (up to a certain distance). The collected local topology supports a randomized local leader election, in which a non leader processor that does not identify a leader within a certain distance x tries to convert itself to a leader. Leaders within distance x from each other are eliminated, until there are no leaders that are within distance x or less from each other. Higher level partitions, using larger distances and overlay network abstraction between leaders, are constructed in a similar way.

In asynchronous systems, our clustering algorithm uses (for each processor) a (local) self-stabilizing snapshot algorithm for obtaining local synchronization of actions.

Self-stabilizing snapshots. We present a snap-stabilizing (e.g., [6]) snapshot algorithm for distributed systems, that uses message passing with *bounded link capacity*, in which a spanning tree is distributively defined. Our snapshot algorithm is designed for a message passing system in which any initial state of link contents is considered and in which the possibility of messages overflow (due to sending a message through a full link) is incorporated into the model.

Our snapshot algorithm can also be applied to systems with a general communication graph in which a rooted spanning tree is distributively defined by another self-stabilizing algorithm. The spanning tree may be an output of a self-stabilizing (BFS) rooted tree construction algorithm. In this case, however, we obtain only *on-demand stabilization* rather than snap-stabilization. On-demand stabilization ensures that regardless of the number of new requests (for snapshots), the system reaches a state, such that eventually any new request results in a correct output (snapshot). In other words, stabilization does not rely on repeated invocations of new (snapshot) requests. Our on-demand self-stabilizing snapshot algorithm serves us as a basic building block in order to obtain our hierarchical snapshot schemes.

Overlay network based snapshot. We suggest an approach for hierarchical snapshot based on an (fifo preserving) overlay network abstraction. We enable each subsystem to perform an independent snapshot, and further enable each level of the hierarchy to perform a local snapshot. We suggest the use of overlay communication links which "directly" connect leaders of clusters. It is worthwhile noting that an (fifo) overlay network link may be in fact a path of physical links. It is also evident that the communication over an overlay link is much faster than the sum of the single hop communication links that implement the overlay link.[1]

Leaders of subsystems are defined, and the communication between processors in different subsystems traverses the overlay communication links between the leaders of the subsystems. Thus, there is no need for recording the messages over physical links between subsystems unless they are part of an overlay communication link. When a snapshot is invoked by a leader of a subsystem (possibly due to a request forwarded to the leader by another processor), the leader uses the overlay network to notify (send snapshot markers to) the leaders of the subsystems that belong to its subsystem. These leaders, in turn, are responsible for performing a snapshot in their subsystem in the same manner.

[1] In some cases, preassigned frequencies or/and supporting switching hardware can be used. e.g., MPLS [5].

Discussion concerning overlay network architecture. We assume the existence of communication switches that are reconfigurable by (commands of) our algorithm. Our approach is layered; the first layer is based on traditional point to point neighboring communication, where communication is between processors that are directly connected by physical communication mean. The output of this layer (which is the local topology of each processor) is used to configure overlay links, using the capabilities of the switches. An analogous procedure is implemented for higher levels in the hierarchy defining new overlay links using wider topology knowledge. We assume that the bandwidth of a physical communication link is sufficient for implementing of all the overlay links that this link participate in implementing (this number is typically small, and is always less than the number of possible source–destination for the overlay links).

One may wish to employ our algorithms to system that does not consist of the above programmable switches. In other words, to provide an abstraction of overlay links in software. To ensure message delivery in such a case, one may need local buffers in each intermediate processor along the overlay path. Each processor may maintain a message buffer for each outgoing edge. The buffer will hold a "bucket" for each overlay path which traverses the corresponding link (again, this number is typically small). Each bucket holds the last message received which is associated with the bucket's path and did not yet traverse the attached link. The processor will send the contents of the buckets repeatedly and fairly (say, simultaneously using high bandwidth). Thus, ensuring eventual delivery (fairness) and FIFO ordering.

As assumed in the scope of overlay communication networks, processing (of higher level protocol-stack are avoided and) is done only at the end-points of the communication, therefore the delay is still assumed to be one time unit. End-to-end ARQ stabilization can be analyzed in the way suggested in [7], resulting in a constant time as well (twice the number of round trip time between the overlay endpoints).

1.2. Related work

Self-organization. In recent years, the concept of self-organization has been widely mentioned in the scope of distributed computing and peer to peer networks. Many works have claimed being self-organizing, but a mere fraction of these works also tries to give a specific definition of what self-organization really is. In [2] a framework for self-organization is proposed, including formal definitions of the self-organization concept and complementary proof techniques which can be used to prove that algorithms are indeed self-organizing. Each algorithm is required to have an associated evaluation criterion, which operates on the immediate neighborhood of a process. This evaluation criterion does not take into account the influence of other local neighbors, say those that are within a constant distance.

Fault containment. Fault containment, using persistent bits, voting on replicated bits (usually for non reactive systems) is another way of addressing locality (e.g., [19,14,1,3]). The idea is to repair transient faults starting from a safe global system configuration. In such a case, it is possible (unlike in the case of topology changes) to change the state of the affected processors back to the state prior to the fault. In this context, our algorithm is self-stabilizing and when started in a safe configuration can handle k transient faults *as well as topology changes* occurring approximately at the same time, in expected $O(\log k)$ rounds. Moreover, our scheme is the first to support many core distributed tasks, such as self-stabilizing leader election algorithm and snapshots algorithms in $O(\log^2 n)$ expected rounds.

Cluster and hierarchy construction. Self-stabilizing and self-healing constructions of hierarchies, in the domain of sensor networks, appear in [28]. The authors divide the plane into hexagonal cells. In each cell a *head* that corresponds with a cluster leader is elected. The existence of a unique processor, the *big node*, which acts as an initiator is assumed. The big node determines the center of the first hexagon, fixating the location of its own cluster. The big node elects heads in adjacent hexagonal cells which will subsequently elect heads in their adjacent cells. The time complexity of this algorithm is obviously proportional to the diameter of the communication graph. Our algorithm does not assume a leader and converges within $O(\log n)$ expected number of rounds and reacts to dynamic changes locally.

Our clustering algorithm is in fact a maximal independent set algorithm. A classical maximal independent set algorithm is presented in [24]. The algorithm is designed for a synchronous system and converges (from a pre-defined initial state) within $O(\log n)$ expected convergence time. Our algorithm is designed for asynchronous systems, is self-stabilizing and self-organizing and converges within expected $O(\log n)$ rounds for constant degree graphs.

A recent work by Wattenhofer and Moscibroda [25] presents an algorithm for computing a maximal independent set in radio networks. The system model is fundamentally different from the one presented here: Processors can broadcast their messages asynchronously, but no collusion detection mechanism is provided. The algorithm presented converges in (expected) polylogarithmic time, and processors which join the algorithm are promised to be covered in (expected) polylogarithmic time.

In [21], the authors present lower bounds on distributed approximation algorithms for the minimum vertex cover problem. Their bounds can also be applied to the maximum independent set problem. We do not seek a maximum independent set, and our algorithm defines a maximal independent set.

Other approaches for distributively defining maximal independent sets in bounded degree graphs appear, for example, in [20] and in [15]. The algorithms presented usually define a maximal independent set in $O(\log^* n)$ rounds. However, a synchronized environment is assumed and is heavily relied upon; for example, in [15] the authors first define a coloring of the graph, using a bounded number of colors. The colors are then used to define a maximal independent set iteratively, by first choosing all the processors colored with the lowest color, removing all of their neighboring processors and repeating the

process with the next color. Unfortunately, these algorithms do not fit asynchronous systems, nor are designed to tolerate faults and dynamic changes gracefully.

Applications of hierarchy in the self-stabilization domain are described in [13]. The authors argue that the hierarchical construction can be used to shorten the convergence time of various self-stabilizing distributed algorithms. As an example, the authors present an application to spanning tree construction. However, the authors do not present an algorithm for defining the hierarchy but assume it is defined beforehand.

Distributed snapshots. The first *distributed snapshot* algorithm was introduced in [4]. The authors describe a distributed algorithm for collecting the states of processors and the states of links such that a global state of the system, called the system snapshot, that has special properties is obtained. Namely, the obtained system snapshot can be reached by an execution that starts in the system state in which the snapshot algorithm was initiated. Moreover, there is an execution that starts from the obtained system snapshot and reaches the system state in which the snapshot algorithm terminated. Therefore, the system snapshot is a global state that can be used to detect stable properties. For example, if there is a deadlock in the global state recorded by the snapshot algorithm, then we may conclude that there is a deadlock in the system.

The snapshot algorithm is defined for message passing system, and is based on special messages called *markers*, which are used to partially order processors' actions. The algorithm is based on rules, which state for each processors, p, the steps p must take each time p receives a marker m on a communication link l: if m is the first marker p received, p records p's local state and immediately sends markers on all of p's outgoing links. Moreover, p records the state of l as empty. If m is not the first marker p received, p records the state of l as the list of messages received from l following the first marker p received. When p received a marker from each incoming link, p publishes its portion of the snapshot which consists of p's recorded state and the state of all the links adjacent to p. The combined published portions of all the processors form the global snapshot. The algorithm is initialized by one or more processors sending markers to themselves and terminates when each processor received markers on all of its adjacent links.

Self-stabilizing snapshot. A self-stabilizing snapshot algorithm was first introduced in [17], where repeated invocations of snapshots are used to ensure stabilization of a non-stabilizing algorithm. When the obtained snapshot indicates an inconsistent system configuration, a reset is invoked. The stabilization of the snapshot itself is based on its repeated invocation. We present an on-demand self-stabilizing snapshot that does not rely on repeated invocations and, in fact, reaches a safe configuration also in cases in which snapshot invocations cease as well. Following [17], several works have studied ways of achieving efficient snapshots in different models e.g., message passing, bounded links message passing and shared memory [27,1,6].

In [27], the author takes a different approach to self-stabilizing snapshots. A common counter is shared among processors and is used to number markers of the snapshot algorithm. Processors only participate in snapshots which match their counter value. In order to obtain self-stabilization, the counter is reset using a self-stabilizing reset algorithm. The system settings do consider links of bounded capacity, but assume that this capacity is never reached. Our algorithm handles link overflows gracefully.

A different approach for the snapshot task is taken by using a snap-stabilizing propagation of information with feedback (PIF) algorithm [6]. In [6], the authors present a snap-stabilizer — a tool that converts any given shared memory algorithm to a snap-stabilizing one by using a technique similar to the one in [17]. The snapshot algorithm uses snap-stabilizing PIF. Shared communication registers are used in [6] for communication among processors. We consider message passing systems. It is worthwhile noting that the conversion of a shared memory algorithm to message passing suggested in [11,9] does not preserve the snap-stabilization property, at least when randomization is not used.

Dynamic graph algorithms. Extensive research on distributed dynamic algorithms appeared in the literature (e.g., [12] and the references therein). Still, our algorithm is the first self-stabilizing and self-organizing distributed (graph) algorithm. Another related aspect of our work is related to dynamic (graph) data structures (e.g., [16] and the reference therein). We achieve a committing time (logarithmic and polylogarithmic) in (fault tolerance) distributed settings for an important class of graphs.

Our contribution. We define the self-organization property to capture locality and dynamicity. We present a clustering algorithm (in fact, a distributed maximal independent set algorithm) which is both self-stabilizing and self-organizing. To realize the clustering algorithm in an asynchronous system we present a scheme of local synchronization, achieved by using a local snapshot protocol. We employ the aforementioned clustering algorithm to define a graph hierarchy which can be used to convert any distributed task to be self-stabilizing incurring only a sublinear time overhead.

Paper organization. In Section 2 we present the system model and in Section 3 the basic on-demand snapshot algorithm. Hierarchy construction schemes are described in Section 4. The hierarchical snapshot algorithm is presented in Section 5. Extensions and concluding remarks appear in Section 6.

2. System model

The *system* consists of *n processors*, denoted by $p_1, p_2, \ldots, p_n$. The processors are connected by *communication links*. Each *processor* is modeled by a state machine that can send and receive *frames* (or low level messages) to/from a subset of the processors. We use a uni-directed communication graph $G = (V, E)$ to represent the system, where each processor p_i is represented by a vertex $v_i \in V$ and each communication link used for transferring frames from p_i to p_j is represented by an edge $(i, j) \in E$. We further assume that the existence of the edge $(i, j) \in E$ implies the existence of an opposite directed

edge $(j, i) \in E$ and that the number of edges attached to a processor is bounded by a constant. We define the *dist* of two processors p and q, $dist(p, q)$, as the length of the shortest path between p and q in the graph. For a processor p and a constant x, we denote $f_p(x)$ as the number of processor q such that $dist(p, q) \leq x$. We further define $f_G(x)$ (or just $f(x)$ where G is clear from the context) as the maximal $f_p(x)$ over all processors p in the graph.

Processors may join and leave the system at any time. Similarly, links may spontaneously fail and recover. We model processors' join and leave as the addition or removal of all of their links from the system. We assume that processors may detect such topological changes in a timely fashion (e.g., by observing voltage levels of the underlying physical layer). In the context of self-organization the pattern and the sequence of topology changes influence the convergence time. We require that following a single topological change at most $o(s(n))$ rounds are needed for stabilization. In case k topological changes occurs together or in a sequence, such that any two consecutive changes among these k changes took place within $o(s(n))$ asynchronous rounds, and within $o(s(n))$ distance apart, then the stabilization time is bounded by $\min\{k \cdot o(s(n)), s(n)\}$ rounds. Note that, any (non constant) number of changes occurring approximately simultaneously in the graph, but in distance of at least $o(s(n))$ from each other, will require only $o(s(n))$ rounds to stabilize.

We assume a class of graphs for which a correlation exists between the number of edges along a shortest path and the geographical distance of the path's end-points.

The system is asynchronous, meaning that there is no correlation between the non constant rate of steps taken by the processors. We assume that the capacity of the communication channels (equivalently the number of items in the fifo queues that represent the links) is bounded, by the constant lc. Whenever a processor p_i sends a frame to a neighbor p_j, when the link (i, j) already contains lc frames, we assume that one of the frames (not necessarily the new one) is lost while the fifo order of the rest of the frames is preserved. In fact, since frames can always be lost, we restrict the pattern of frame loss steps to be such that if frames are sent infinitely often, frames are also received infinitely often.

We further abstract the activity of communication links by assuming an underline snap-stabilizing ARQ data link algorithm that transfers frames in order to ensure that high level *messages* transfer respects the following: (1) messages sent from p_i to p_j are received by p_j in a finite (but yet unbounded) time (2) and message delivery respects the exactly once delivery and fifo ordering policies. We note that the ARQ algorithm performed on one link of a processor p_i does not block the receive operations (and corresponding steps) from the links attached to p_i. We assume that eventually when p_i sends a message m to p_j (and p_i does not send further messages), p_i receives acknowledgment for m after p_j received m.

We use the term overlay edge to denote a path of edges that connects two processors in the system. When the path is predefined and fixed, it acts as a virtual link in which (practically) no processing is required by intermediate processors in order to forward the frame from source to destination. We allow processors to define and use, on the fly, overlay edges to other processors, when the underlying path is known. We regard the time it takes a frame to traverse such an overlay link as the time for traversing a link that directly connects two neighboring processors. We assume these overlay edges preserve FIFO ordering of frames between processors and maintain the assumption that a frame which is infinitely often sent is infinitely often received.

A configuration c of the system is a tuple $c = (S, L)$; S is a vector of states, $\langle s_1, s_2, \ldots s_n \rangle$, where the state s_i is a state of processor p_i; L is a vector of *link states* $\langle l_{1,2}, l_{1,3}, \ldots, l_{2,1}, l_{2,3} \cdots \rangle$. A *link* $l_{i,j}$ is modeled by a fifo queue of frames that are waiting to be received by p_j and the contents of the queue is the state of the link. Whenever p_i sends a frame f to p_j, f is enqueued in $l_{i,j}$. Also, whenever p_j receives a frame f from p_i, f is dequeued from $l_{i,j}$. A processor changes its state according to its transition function (or program). A transition of processor p_i from a state s_j to state s_k is called an *atomic step* (or simply a step) and is denoted by a. A step a consists of local computation and of either a single send or a single receive operation.

We model our system using the interleaving model. An *execution* is a sequence of global configurations and *steps*, $\mathcal{E} = \{c_0, a_0, c_1, a_1, \ldots\}$, so that the configuration c_i is reached from c_{i-1} by a step a_i of one processor p_j. The states changed in c_i, due to a_i, are the one of p_j (which is changed according to the transition function of p_j) and possibly that of a link attached to p_j. The content of a link state is changed when p_j sends or receives a frame during a_i. An execution $\mathcal{E}$ is *fair* if every processor executes a step infinitely often in $\mathcal{E}$ and each link respects the bounded capacity loss pattern. In the scope of self-stabilization we consider executions that are started in an arbitrary initial configuration.

A *task* is defined by a set of executions called *legal executions* and denoted *LE*. A configuration c is a *safe configuration* for a system and a task *LE* if every fair execution that starts in c is in *LE*. A system is self-stabilizing for a task *LE* if every infinite execution reaches a safe configuration in relation to *LE*. We sometimes use the term "the algorithm stabilizes" to note that the algorithm has reached a safe configuration with regards to the legal execution of the corresponding task.

In some cases, we would like to define *processes* executed by the processors so that each processor executes steps for several processes. Consider the case in which each processor p_i executes two processes p_i^1 and p_i^2. Assume further, that a process p_i^1 can communicate directly with a (neighboring) process p_j^1 residing in a neighboring processor. The transition function of p_i^1 is defined by the state s_i^1 of p_i^1 and the messages received from a neighboring processes p_j^1. The transition function of p_i^2 is defined by the state of p_i^1 and p_i^2 and the messages sent by neighboring processes p_j^2. The definition of configuration for the multi-processes case is defined by a vector of state $\langle s_1^1, s_2^1, \ldots \rangle$ for the state vector of the *first layer* processes and a vector $\langle l_{1,2}^1, l_{1,3}^1, \ldots \rangle$ of the link states of the *first layer*, while the later is composed of the queues associated with the links, restricted to the messages sent by processes in layer one, p_i^1. The layer's definition allows us to separate the snapshot protocol activity (in the lowest layer) from the original system (upper layers) that is the subject of the snapshot.

A multi-process fair execution is a fair execution in which every *process* executes a step infinitely often (in the sequel we use the term fair execution for multi-process fair execution).

The *snapshot task* $\mathcal{S}$ for a system is defined by a set of executions $\mathcal{E}_{\mathcal{S}}$ started in an arbitrary configuration, so that if a snapshot starts in an atomic step a_r, there is a configuration c_s, that follows a_r, in which a processor receives a global snapshot *gs*. Moreover, assuming r is minimal, there exists an execution of processes in level one that starts immediately before a_r, reaches *gs* and then continues to the configuration of level one in c_s.

We use the notion of *asynchronous rounds* to measure the time complexity of an algorithm. The first *asynchronous round* in execution $\mathcal{E}$ is the shortest prefix of $\mathcal{E}$ in which each processor (or process) communicates with all of its neighbors (either through a directly connecting communication link or through an overlay edge). The second asynchronous round in $\mathcal{E}$ is the first asynchronous round of the suffix of $\mathcal{E}$ that immediately follows the first asynchronous round in $\mathcal{E}$. The time complexity of an algorithm is the number of asynchronous rounds (or simply rounds) that are required to achieve the task of the algorithm.

3. On-demand (snap-)stabilizing message passing (tree-)snapshot algorithm

In this section, we present the first snap-stabilizing snapshot for message passing systems. We do not require repeated invocations of the snapshot algorithm in order to stabilize, in contrast to the assumption needed in order to employ the snapshot algorithm of [17]. A snap-stabilizing snapshot algorithm for the shared memory system is presented in [6]. In the context of self-stabilization, message passing systems introduce additional intricacy due to unknown messages in transient in the arbitrary first configuration from which the system should converge to a legal behavior [9].

When designing our snapshot algorithm, our starting point is the unbounded snapshot algorithm presented in [17] and the snap-stabilizing algorithm presented in [6], which we modify to a bounded message passing snap-stabilizing algorithm. Namely, we ensure that any new request for a snapshot will result in a correct snapshot. This requirement differs from the one presented in [17] where snapshots must be continuously and infinitely often invoked. In our case, the algorithm is ready for future requests even when no snapshot requests are made.

The algorithm is designed for a system in which a rooted spanning tree is distributively defined. It is based on performing two consecutive tree-PIFs (propagation of information with feedback using a spanning tree) and then employing the original snapshot algorithm of [4]. Each PIF uses the rooted tree in order to propagate a command (*initialize* and then *prepare*) and receive feedback on the completion of the propagation (of the initialize and prepare commands, respectively). A processor that receives a command from its parent, propagates it to its children and also "cleans" the non-tree edges attached to it. Once a processor p receives an acknowledgment from all its children that their subtree received the command and once p finishes cleaning the attached non-tree links, p sends an acknowledgment to its parent regarding the completion of the command propagation. Both tree-PIFs are completed within $O(d)$ rounds (assuming a BFS tree is used), where d is the diameter of the network. When the first (initialize) tree-PIF is completed, no marker of previous incarnations of the snapshot algorithm is present in the system and processors disregard all incoming snapshot markers. After the second (prepare) tree-PIF is completed, processors do not ignore markers and the root may then initiate the original snapshot algorithm of [4].

To guarantee snap-stabilization we have to ensure that when the root starts a tree-PIF and then receives an indication from its children regarding completion, the system's configuration is indeed the desired one - namely, a configuration in which all nodes are instructed by the propagated command. The technique used to achieve the above is based on a method of ensuring the *happened before* (see [22]) relation, using a snap-stabilizing data link algorithm which is specifically designed for bounded capacity links (Fig. 2). When a processor p would like to pass a command to a neighbor q, p repeatedly sends frames with a label i until p receives a frame with label i from q. Then p repeatedly sends frames with label $i+1 \pmod{2 \cdot lc+1}$ to q until a frame with the new label is received from q and so on until p sends $2 \cdot lc + 1$ distinct labels. In each frame q sends the last *local synchronization color* q received from p. Thus, when p receives a frame with the last label among the set of the distinct $2 \cdot lc + 1$ labels, p knows the current local synchronization color known to q and sends frames with a different local synchronization color together with the global command (initialize or prepare) that p would like to pass to q. Following that, q identifies the new local synchronization color and invokes the global command.

To simplify our presentation we use a self-stabilizing version of the aforementioned *frame communication algorithm*. The self-stabilizing frame communication algorithm is used to send *control messages* between neighboring processors. Each control message is either piggy backed on messages sent by the original algorithm (the snapshot subject) or sent independently (as part of a frame). Each processor p maintains 3 arrays: *next*, *current* and *last*. Each array has an entry for each neighbor of p. $next[q]$ is the entry in which the next value that p is about to send to q is stored. p may decide to send a different value to q before $next[q]$ is sent. In such a case, the value in $next[q]$ is overwritten. $current[q]$ holds the data that p is currently sending to q. $last[q]$ contains the last acknowledged data that p sent to q along with the actual acknowledgment of q. We note, that transforming the self-stabilizing version presented in Fig. 2 into a snap-stabilizing one can be achieved by iterating the sending operation $2 \cdot lc + 1$ times. Correctness is trivially preserved and the conversion adds a constant amount of time to each send operation which can be considered $O(1)$ for a time complexity measure.

We now describe the way $next[q]$, $current[q]$ and $last[q]$ are accessed. We use Figs. 1 and 2 in our description. For each frame arriving from q, p checks whether the frame contains an acknowledgment (Fig. 2 line 2). The acknowledgment should also be numbered with the current number that p is expecting to receive from q. When an acknowledgment with the current number arrives, p "advances" the values $next[q]$ to $current[q]$, and $current[q]$ to $last[q]$. In more details, $last[q]$ is assigned by

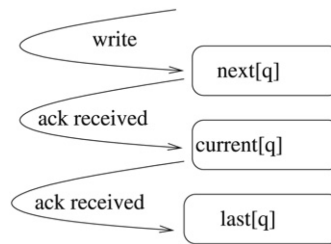

Fig. 1. Frame communication algorithm - data flow.

current[*q*], *current*[*q*] is assigned by *next*[*q*], and *next*[*q*].*val* is assigned by null, while *next*[*q*].*num* is assigned by the next control number (Fig. 2, line 8).

If the frame arriving is not an acknowledgment, *p* first sends an acknowledgment to *q* (line 10). Afterwards, if *q* is the parent of *p* and if *q* sent a control message with a different color than *p*'s color, *p* changes state accordingly (line 13). Alternatively, if *p* is the parent of *q*, and the frame contains a DONE message, *p* updates the data structure which denotes which child has finished the current initialization phase.

At the end of the frame communication algorithm, *p* passes the encapsulated message to the algorithm subject to the snapshot. Equivalently, in line 21, each message destined to *q* is encapsulated in a frame, holding *current*[*q*].

The self-stabilizing snapshot initialization algorithm is responsible for cleaning old markers from the communication links. Roughly speaking, there are two phases, in each of which the root instructs each child, *q*, to color itself with a color that is different from the color that *q* is currently colored by. Each processor *q* repeats the same procedure with its tree children and sends an acknowledgment to its parent once it is done. Following the initialization, processors do not participate in snapshots. Later on, when the root receives DONE messages from each of its children, the root starts the prepare phase. When the second phase is completed, the processors may start participating in a snapshot.

The code of the initialization-prepare algorithm appears in Figs. 3 and 4. We use a convention inspired by the *guarded commands* notation for representing the program of a processor. The program in Fig. 3 is composed of six guarded commands. Each *guard* is a predicate. A guard is *enabled* if, and only if, the predicate is evaluated as true. Each *command* is a finite set of instructions that a processor must take when the corresponding guard is enabled. We assume that the guards of a certain processor are scheduled by an *internal scheduler* which repeatedly chooses to execute the commands of an enabled guard. Furthermore, we assume that the internal scheduler ensures that a guard that is infinitely often enabled is executed infinitely often.

In general, the initialization-prepare algorithm task uses phases identified by processor states to coordinate the operation of the processors. The algorithm has two main phases (INITIALIZE, PREPARE) each of which has a few states associated with it. In particular, the initialization phase is composed of state changes according to the following order: INITIALIZE, SYNC_INITIALIZE, PROPAGATE_INITIALIZE, CHILDREN_INITIALIZE, FINISH_INITIALIZE and finally the DONE state.

The first guard of the initialization-prepare algorithm (Fig. 3, line 1) describes the first actions a processor (*p*) takes when starting a new phase (either INITIALIZE or PREPARE). First, *p* cleans the links connecting it to its neighbors. The cleaning is achieved by sending a PROBE message on each link. Each neighbor *q*, when receiving a PROBE message, acknowledges receiving the message and attaches to the acknowledgment *q*'s current synchronization color (Fig. 2, line 10). After sending all PROBE messages, *p* will change state to the SYNC state, e.g., if *p*'s state was INITIALIZE, it will change state to SYNC_INITIALIZE (line 8). Moreover, if *p* was in the INITIALIZE state, then before changing state it will set the *ignore_markers* flag to true (line 7).

The second guard appears in line 10. If *p* is in the SYNC_INITIALIZE (or SYNC_PREPARE) state and receives acknowledgments from all of its neighbors, line 11 ensures that *p* will change state to PROPAGATE_INITIALIZE or PROPAGATE_PREPARE respectively. Line 13 describes the actions *p* must take in the PROPAGATE states. Essentially, *p* propagates its *phase* state to its children (lines 14–19). We define the phase state as INITIALIZE if *p* is in the initialize phase, and PREPARE if *p* is in the prepare phase. For example, if *q* is in the PROPAGATE_INITIALIZE state, the phase state for *p* is INITIALIZE. Similarly, if *p* is in the PROPAGATE_PREPARE state, the phase state for *p* is PREPARE. The propagation is achieved by sending an appropriate command to each child. The command contains a color that is different from the last color each child had already sent (line 19). This ensures that the children identify the command as a new command and change state accordingly. Finally, *p* changes state to CHILDREN_INITIALIZE (or CHILDREN_PREPARE).

The third guard appears in line 23. This guard ensures that *p* will wait for each child to acknowledge the command. After each child acknowledges the command, *p* must wait for its children to finish propagating the command to their subtrees. To this end, *p* utilizes the *done*[] array. When a processor is in the DONE state, it repeatedly sends DONE messages to its parent. To make sure that *p* considers only relevant DONE messages, *p* must first initialize the *done*[] array with *false* (line 25). *p* will then change state to the FINISH state (with the appropriate suffix, according to *p*'s phase state).

Once all of *p*'s children finish synchronizing their sub-trees, they will send a DONE message to *p*. This will cause the guard in line 29 to be enabled. At this point, *p* will have finished synchronizing its subtree and can proceed to the DONE state. However, before changing state, if *p* is in the PREPARE state, it will change *ignore_markers* to *false*. From this point on, *p* is ready to participate in a snapshot.

```
1 new frame(num,val,in_message) arrived from q:
2   if val = (ack, color) ∧ num = current[q].num then
3   |   current[q].color ← color
4   |   last[q] ← current[q]
5   |   current[q] ← next[q]
6   |   next[q] ← nil
7   |   next[q].num ← current[q].num+
8   |       1 (mod 2 · lc + 1)
9   else
10  |   SendFrame(num,(ack,color))
11  |   if q = parent ∧ val = (new_color, command)∧
12  |       new_color! = color then
13  |       state ← command
14  |       color ← new_color
15  |   if q ∈ Children ∧ val = DONE then
16  |       done[q] = true
17  |   fi
18  fi
19  pass in_message upwards in the protocol stack
20 end

21  SendFrame(current[p].num, current[p].val, message)
```

Fig. 2. Frame communication with a neighbor q.

The last guard, which is in line 34, ensures that if p is in the DONE state, p will repeatedly send DONE messages to its parent (line 35).

Correctness proof. We will first show that the data link algorithm stabilizes. The proof which is already a folklore, is for a particular pair of processors, for example p_i and p_j, where p_i is the sender and p_j is the receiver.

Lemma 1. *In every fair execution $\mathcal{E}$, assume p_i assigns a new value to $current_i[p_j]$ in a global configuration c_l. Then, there exists a global configuration c_k, such that an acknowledgment from p_j arrived at p_i and $k > l$. Furthermore, eventually the* happened before *relation holds between the atomic steps a_k in which the assignment of a new value x to $current_i[p_i]$ is executed, a later atomic step a_l in which a frame with x is received by p_j and an atomic step a_m in which the acknowledgment regarding the receipt of x in p_j is received by p_i.*

Proof. Since each frame numbered with *num* is sent repeatedly by p_i, an acknowledgment with *num* will eventually arrive from p_j. Hence, p_i will infinitely often change the frame number in a round robin fashion (Fig. 2, line 8). Since the link capacity is bounded by lc and p_i increments the frame numbers modulo $2 \cdot lc + 1$, a value y which is not present in the first arbitrary configuration in either the link (i, j) or (j, i) will be chosen. From this point on, it is obvious that our claim holds, since p_i will only accept acknowledgments for y.

Considering the snap-stabilizing version, an analogous proof can be derived to show that the data link algorithm is indeed snap-stabilizing. ■

Lemma 2. *In every fair execution $\mathcal{E}$, eventually after a processor p_i executes lines 3–4 in Fig. 3 (denoted* writing*) for a particular neighboring processor p_j and assuming no consecutive write of p_i to p_j takes place, an acknowledgment will arrive at p_i. Immediately after the atomic step in which the acknowledgment arrives, a configuration in which $last_i[p_j]$ is equal to $ack_pending_i[p_j]$ will be reached (Fig. 4).*

Proof. Since no writes occur and since p_i repeatedly sends frames with $current_i[p_j]$ an acknowledgment for $current_i[p_j]$ will eventually arrive. Since we assumed no writes occurred, p_i will assign $next_i[p_j]$ to $current_i[p_j]$, according to Fig. 2 lines 3–5. Using Lemma 1 we know that an acknowledgment will eventually arrive after p_j receives $current_i[p_i]$. Hence, and again according to lines 3–5, p_i will assign $current_i[p_j]$ to $last_i[p_j]$. We can conclude the correctness of the Lemma from the fact that the value that was originally present at $next_i[p_j]$ was copied to $ack_pending_i[p_j]$ and eventually to $last_i[p_j]$. ■

For the following lemmas, we assume that the data link algorithm used is the snap-stabilizing version discussed earlier. We wish to draw the readers' attention to the fact that the correctness of the on-demand version also holds if the following condition is met: the self-stabilizing data link has stabilized – a step that ensures that whenever an acknowledgment arrives for a frame sent by p_i, p_j will have received the message. Overall correctness is further ensured by the fair composition technique ([9], chapter 2.7).

Lemma 3. *In every fair execution, if a processor p_i is in state* INITIALIZE *at a configuration c_j and does not receive any command from its parent to change its state, then there exists a configuration c_k, $k > j$, such that the following claim holds for each processor q in the subtree of p_i (including p_i): there exists a series of configurations (after c_j), in which q changed state from* INITIALIZE *to* SYNC_INITIALIZE *to* PROPAGATE_INITIALIZE *to* CHILDREN_INITIALIZE *to* FINISH_INITIALIZE *and finally to* DONE. *Furthermore, q stays in the* DONE *state in all subsequent configurations, after (and including) c_k.*

```
Predicate answered(q) ≡ ack_pending[q].num = last[q].num

1 state ∈ {INITIALIZE, PREPARE} :
2   foreach q ∈ Neighbors do
/* this is actually the write */
3   |   ack_pending[q] ← next[q].num
4   |   next[q].val ← PROBE
5   done
6   if state = INITIALIZE then
7       ignore_markers ← true
/* change state to SYNC.INITIALIZE becomes SYNC_INITIALIZE */
8   state ← SYNC_state
9 end

10 ∀q ∈ Neighbors| answered(q)∧
    (state = SYNC_INITIALIZE ∨state = SYNC_PREPARE):
/* SYNC_INITIALIZE becomes PROPAGATE_INITIALIZE */
11  state ←PROPAGATE_base_state
12 end

13 (state = PROPAGATE_INITIALIZE∨state =PROPAGATE_PREPARE) :
14  foreach q ∈ Children do
15  |   ack_pending[q] ← next[q].num
16  |   if state = PROPAGATE_INITIALIZE then
17  |       child_command ← INITIALIZE
18  |   else child_command ← PREPARE
19  |   next[q].val ← (last[q].color‾, child_command)
20  done
/* PROPAGATE_INITIALIZE becomes CHILDREN_INITIALIZE */
21  state ←CHILDREN_base_state
22 end

23 ∀q ∈ Children| answered(q) ∧ (state = CHILDREN_INITIALIZE∨
    state =CHILDREN_PREPARE):
24  foreach q ∈ Children do
25      done[q] ← false
26  done
27  state ← FINISH_base_state
28 end

29 ∀q ∈ Children| answered(q) ∧ done[q] = true∧
    (state = FINISH_INITIALIZE ∨state =FINISH_PREPARE):
30  if state = FINISH_PREPARE  then
31      ignore_markers ← false
32  state = DONE
33 end

34 state = DONE:
35  next[parent].val ←DONE
36 end
```

Fig. 3. Initialize-Prepare algorithm for a processor p.

```
1 start a new snapshot:
2   state ← INITIALIZE

3 state = PREPARE_DONE:
4   the root is ready to start a new snapshot.
```

Fig. 4. Root rules for initiating a new snapshot.

Proof. The proof is by induction on h, the height of processors in the tree. For $h = 0$ we have a leaf processor p_i. Assume p_i changed state to INITIALIZE at c_j. According to the initialization algorithm (Fig. 3), the only guard enabled is the guard in line 1. Since this guard is the only one enabled, it will eventually get executed. p_i then writes a PROBE message to each of its neighbors and changes state to SYNC_INITIALIZE (lines 1–9). Since we assume that p_i receives no command from its parent to change state, no guard is enabled by default. According to Lemma 2, the guard in line 10 is the only guard which will eventually be enabled. p_i then changes state to PROPAGATE_INITIALIZE. The next enabled guard is only the guard in line 13. Since p_i is a leaf, p_i will immediately change state to CHILDREN_INITIALIZE and then to FINISH_INITIALIZE. The only guard enabled now is the guard in line 29, since p_i does not need to wait for an answer from any child. Hence, p_i will change state to DONE. Marking the last configuration as c_k concludes the proof for the base case.

Now, let p_i be a processor of height greater than 0. Assume p_i changed state to INITIALIZE at c_j and no further commands arrive from p_i's parent. The only enabled guard in c_j appears in line 1 and will eventually be executed. p_i will then send a

PROBE message to each of its neighbors and change state to SYNC_INITIALIZE. No guard will be enabled until each neighbor replies. However, Lemma 2 ensures that a reply will eventually arrive. Hence, the guard in line 10 will eventually be enabled and executed. p_i will then change state to PROPAGATE_INITIALIZE. The guard in line 13 will now be enabled, and executed. p_i will send an INITIALIZE command to each of its tree children, and change state to the CHILDREN_INITIALIZE state. Again, from Lemmas 1 and 2, eventually the guard in line 23 will be enabled and p_i will reinitialize the *done* array for each child to hold *false*. Since Lemmas 1 and 2 ensure that p_i gets a reply for the PROBE message from each child that is sent **after** each child receives the aforementioned PROBE message, p_i will send each child a color different to the one this child currently holds. Consequently, each child will enter the INITIALIZE state. As a result, it is ensured that no old DONE messages exist on either channel directed at p_i, since each child has changed state to INITIALIZE before sending the acknowledgment regarding the INITIALIZE command to p_i.

Using the induction assumption and since p_i does not send any more messages to its children (no guard is enabled), we can conclude that the lemma holds for each child of p_i. Moreover, the guard in line 29 will eventually be enabled, since each child is in the DONE state and the only action taken in this state is sending DONE messages to the parent. Hence, p_i will receive DONE messages from each of its children and the guard in line 29 is finally enabled. Now, p_i will enter the DONE state. Based on the induction assumption, each child fulfills the requirements of Lemma 3 and from the proof we get the series of configurations for p_i as required by Lemma 3. Furthermore, p_i does not change its state after reaching the DONE state, unless p_i receives a new command from its parent. ■

Following the proof of Lemma 3 we can deduce a similar argument for the PREPARE state. Using these lemmas, we deduce that once the root changes state to initialize, the whole tree will change state to INITIALIZE and will stop receiving markers (Fig. 3 line 7). It also follows that eventually the root will receive a DONE message from all of its children, ensuring all processors in the tree are in the DONE state. When in this state, we can deduce that no markers exist in any of the channels. Assume the contrary, that between p_i and p_j there exists a marker sent by p_i. p_i would not have been able to send the marker after receiving the INITIALIZE command since after changing state to the INITIALIZE state, p_i ignores all markers and snapshots (Fig. 3 line 7). The only option left for p_i is to have sent the marker before receiving the INITIALIZE command. Since channels are fifo ordered and p_i sent a PROBE message to each neighbor **before** finishing the initialization algorithm, we conclude that no such marker can exist.

After finishing the initialization phase, the root will start the PREPARE phase. After finishing the PREPARE phase, each processor will start to receive markers again (Fig. 3 line 31). Once the root has entered the DONE state, it is ensured that all processors are ready to start a new snapshot and that no old markers exist in the system.

Time complexity:

Lemma 4. *In every fair execution once a processor p assigns* INITIALIZE *to its state, there is an atomic step in the following* $5 \cdot height(p)$ *rounds in which p assigns* DONE *to its state (where* $height(p)$ *is the height of p in the tree).*

Proof. By induction, over the height of a processor in the tree. Let us assume p is a processor of height 1 in the tree (a leaf). Then, according to the snapshot algorithm (Fig. 3), the steps p follows are: sending a probe to all neighbors (time complexity: 1 round (line 1)), waiting for an answer to the probe (time complexity: 1 round (line 10)), propagating the INITIALIZE command to each child (line 13) and finally waiting for a DONE message from each child (line 29). The last two steps are internal events, since p has no tree children and no communication is needed. As a result, an atomic step in which p assigns DONE to its state is executed after 3 rounds.

Let us assume Lemma 4 is correct for all processors of height at most k, for some k. Let p be a processor of height $k + 1$ and assume p assigns INITIALIZE to its state. According to the snapshot algorithm, p must make take the following actions: send a probe to all neighbors (time complexity: 1 round (line 1)), wait for an answer to the probe (time complexity: 1 round (line 10)), propagate the INITIALIZE command to each child (line 13) and finally wait for a DONE message from each child (line 29). The propagation of the INITIALIZE command takes 1 round. Now, since all tree children of p operate in parallel and are of height $k - 1$, in $5 \cdot (k - 1)$ rounds each child will assign DONE to its state (according to the induction assumption) and after another round, the command will be propagated to p. In the following round, p will also set its state to DONE. To conclude, the atomic action in which p assigns DONE to its state is executed in $3 + 5 \cdot (k - 1) + 2 = 5 \cdot k$ rounds. ■

The time complexity of the algorithm, as clearly follows from Lemma 4, is $O(d)$. The root must first assign INITIALIZE to its state and after $5 \cdot height(root)$ rounds the root will receive a DONE message from all of its children. The root will then assign PREPARE to its state, and a similar argument can show that after another $5 \cdot height(root)$ rounds the root will receive a DONE message from all of its children. Overall, the tree will be ready for a new snapshot after at most $10 \cdot height(root) = O(d)$ rounds. The snapshot itself requires an additional $O(d)$ rounds, thus the total number of rounds required for performing a snapshot is $O(d)$.

4. Hierarchical construction schemes

A hierarchical system is represented by a communication graph, $G = (V, E)$ and a hierarchy tree $\mathcal{HT} = (V_h, E_h)$. Each node in $\mathcal{HT}$, l_i, represents a set of nodes in V, called a *subsystem*, so that if l_i and l_j are at the same level of $\mathcal{HT}$, then $l_i \cap l_j = \emptyset$.

Furthermore, if K is a set of nodes at level i of $\mathcal{HT}$, then $\cup_{j\in K} l_j = V$. The nodes of the graph are processors and the edges are their communication channels. We require that each subsystem is a connected component of G.

Next, we present a self-stabilizing and self-organizing algorithm for constructing clusters. In general, the clustering algorithm builds clusters of diameter smaller than a fixed parameter. Furthermore, each cluster is defined by a "native" leader.

4.1. Synchronous cluster construction

The clustering algorithm is based on a self-stabilizing, randomized, synchronous, local leader election algorithm. We assume the existence of a global shared clock. If no such clock exists, a self-stabilizing digital clock synchronization algorithm (e.g., [9]) can be used. However, in such a case the resulting algorithm is not self-organizing. Assume clusters of diameter at most $2 \cdot x$ are desired. All processors will participate in a self-stabilizing update algorithm, up to distance x. At predefined intervals of x clock ticks (which we call a *phase*), all processors will execute the algorithm presented in Fig. 5.

The update algorithm is designed for an asynchronous system. Each processor p holds a table of tuples, each of the form $\langle id_q, dist_q, parent_q\rangle$. Each tuple represents a processor q in the communication graph. id_q is the unique identification of q, $dist_q$ is the minimal distance between p and q and $parent_q$ is the id of a neighboring processor of p, which is the first on a shortest path from p to q. Repeatedly, p combines all the tables of its neighbors and for each of the conflicting tuples (in which the id is the same), p chooses the tuple with the minimal $dist$ (further ties are broken using the $parent$ value). Next, p chooses only entries with $dist = k$, such that there exist entries with $dist = j$ for all $j < k$. All other entries are deleted. Afterwards, p adds 1 to the distance field of every tuple and finally adds the tuple $\langle id_p, 0, nil\rangle$ to form the new table.

We adapt the aforementioned update algorithm to our system in several manners. First, each tuple will hold two extra values, $leader_p$, rtp_p. Next, each processor p continuously sends its table to all neighboring processors. In addition, p maintains an internal array which consists of the most recent topology tables p received from each neighboring processor. The computation of p's topology table is done on the basis of this array. Furthermore, in the validation phase we also delete entries with $dist > x$. Consequently, p's table will reflect its neighborhood up to distance x from p. The correctness of the revised update algorithm is trivially preserved, and the convergence time is $O(x)$ rounds.

Continuing the description of our algorithm, each processor p with $leader_p = true$ first chooses a random temporal identifier rtp for the current phase and uses the tuple $\langle rtp, id\rangle$ as its identifier for the phase. This random choice of an rtp value is used to break the symmetry between processors (for further motivation, see the asynchronous version of the algorithm). The variable $leader_p$ is used to indicate whether p regards itself as a leader or not. The self-stabilizing update algorithm collects the new identifiers and leader variables value within the x clock ticks of the phase. Thus, at the end of the phase, a processor p with $leader_p = true$ checks whether p is the only leader in the area defined by radius x from itself. If there does not exist a processor q with $leader_q = true$ with distance less than x from p, then p is a *stable leader* and does not change state. Otherwise, if $leader_p = false$ and there is no other processor q with $leader_q = true$ within distance x from p, then p assigns $leader_p \leftarrow true$. Last, consider the case in which $leader_p = true$ and there exists another processor q with $leader_q = true$ that is within distance x from p. If p's $\langle rtp, id\rangle$ is larger than q's $\langle rtp, id\rangle$ (first comparing the rtp and breaking symmetry by the use of id) then p assigns $leader_p \leftarrow false$.

The leaders define the cluster structure and since each processor p has at least one leader in its neighborhood, p may choose to join the cluster formed by one of the closest leaders.

To prove that the algorithm stabilizes, we first assume that the update algorithm has stabilized. Hence, at the start of each new phase, each processor holds a consistent table, denoting the processor's neighbors of distance not larger than x. For the proof, we will use a potential function. For each c, a configuration of the system at the end of a phase, define $SL(c)$ to be the number of stable leaders at c.

In the following Lemmas we use the term *synchronous execution* to denote an execution of a synchronous algorithm. For further details see [9].

Lemma 5. *In every synchronous execution, if p is a stable leader in configuration c_i, p will stay a stable leader in every configuration c_j, such that $j > i$.*

Proof. p is a stable leader if, at the end of a phase i, p is the only local leader within a radius of x. Assume, by contradiction, that at the end of phase j, such that $i < j$, p has stopped being a stable leader. If p became a candidate leader, it follows that there exists a processor q, such that $dist(p, q) < x$ and q is also a (either stable or candidate) local leader. Hence, there exists a phase k, such that $i < k < j$ and q has become a leader at the end of phase k. However, this contradicts the fact that p was a leader at the same phase. The second option is for p to assign $leader_p$ by *false* at the end of phase j. This can only be a consequence of p losing (line 14) to another processor q, which was also a leader at phase j, but $\langle rtp_p, id_p\rangle < \langle rtp_q, id_q\rangle$. A similar argument as in the previous case holds and can be used to show that q cannot exist. ■

From Lemma 5 it follows that SL is a monotonically increasing function. The next lemmas will show that if SL cannot be increased any longer, the system has stabilized:

Lemma 6. *In every synchronous execution, if no new stable leader can be added at configuration c_i, either by turning a candidate leader or a regular processor into a stable leader, then each processor p has at least one leader within a distance x in every configuration c_j such that $j \geq i$.*

```
Predicates:
close_leader(p, q) :=
    leader_q ∧ dist(p, q) ≤ x

1 !leader_p ∧ ∃q(close_leader(p, q)):
    /* do nothing (stable). */

2 !leader_p∧!∃q(close_leader(p, q)):
    /* p declares itself a local leader. */
3   leader_p ← true
4   rtp_p ← random()
5   we denote p to be at a candidate state.

6 leader_p∧!∃q(close_leader(p, q)):
    /* declare itself a local leader. */
7   leader_p ← true
8   rtp_p ← random()
9   we denote p to be at a stable state.

10 leader_p ∧ ∃q(close_leader(p, q)):
11 if ⟨rtp_p, id_p⟩ > ⟨rtp_q, id_q⟩ then
        /* p redeclares itself a candidate local leader. */
12      leader_p ← true
13      rtp_p ← random()
14 else
        /* p relinquish local leadership */
15      leader_p ← false
```

Fig. 5. Leader election algorithm for processor *p*.

Proof. The proof is by contradiction. Assume there exists a processor p which has no stable local leader within distance x in configuration c_i and that no stable leader can be added. According to the algorithm, this processor can become a candidate leader and at subsequent phases a stable leader. This contradicts our assumption that no stable leader can be added. ■

Lemma 7. *In every synchronous execution, at the end of each phase i if no new stable leaders were added if and SL can be increased, then there is a positive probability that at the end of phase j, such that $i < j$, there is at least one more stable leader than in phase i.*

Proof. Let us examine the set CL, of all candidate leaders. If $CL = \emptyset$, then in the following phase, at least one processor p will declare itself a local leader (since SL can be increased). If p is a stable leader, then the proof is complete. Otherwise, assume $CL \neq \emptyset$. Let us denote p as the processor with the highest $\langle rtp_p, id_p\rangle$ tuple in CL. During the transition from phase i to phase $i+1$, $leader_p$ will be propagated up to distance x from p. For each q, such that $q \notin CL$ and $dist(p, q) \leq x$, q will be aware that $leader_p$ is *true* and will not change state. Furthermore, each q, such that $q \in CL$ and $dist(p, q) \leq x$, will enter line 14 in the algorithm (since $\langle rtp_p, id_p\rangle > \langle rtp_q, id_q\rangle$) and set $leader_q \leftarrow false$. Thus p will become a stable leader. ■

From Lemma 7 it follows that as long as stable leaders can be added, stable leaders **will** be added. Lemma 6 ensures us that SL is a monotonically increasing function. It is easy to note that SL is also bounded. Therefore, starting from any initial configuration, SL will reach a value from which no new stable leaders can be added. From Lemma 5, it follows that the system has stabilized ■

Lemma 8. *In any synchronous execution, starting with an arbitrary global configuration, the algorithm converges to a* stable state *within $O(\log n)$ expected number of rounds, where a stable state denotes a configuration in which all processors are stable.*

Proof. Define the *neighborhood* of a processor p as the set of all processors q, such that $dist(p, q) \leq x$. Further define $f_p(x) = |neighborhood_p|$ and $f(x) = max_p(f_p(x))$. Note that since the maximal degree of a processor is constant and since x is a constant, $f(x)$ is also a constant. We say that a processor is *stable* if it is either a stable leader or has a stable leader in its neighborhood.

We now bound the probability for a processor p to become a stable leader in $O(x)$ rounds (assuming p has no leader). According to the algorithm, p will set itself a leader and choose a random *rtp*. We would now like to calculate the probability that p chooses a unique *rtp* value, which is larger than any *rtp* value which may be chosen in p's neighborhood. Call this probability P_{success}. Assume each processor chooses *rtp* values uniformly in the range $[1, m]$. Obviously, P_{success} is larger than the probability of the events in which no other processor in p's neighborhood chose the same value as p and p is maximal in

p's neighborhood. The probability that p has the highest *rtp* value, given that no other processor has the same *rtp* value as p, is $\frac{1}{f_p(x)}$. This is due to symmetry considerations. We now calculate a lower bound for P_{success}:

$$P_{\text{success}} > \frac{1}{f_p(x)} \cdot \left(\frac{m-1}{m}\right)^{f_p(x)} > \frac{1}{f(x)} \cdot \left(\frac{m-1}{m}\right)^{f(x)}$$

If we assume that $m = c \cdot f(x)$ for some $c \geq 1$, then we get:

$$P_{\text{success}} > \frac{1}{f(x)} \cdot \left(\frac{c \cdot f(x) - 1}{c \cdot f(x)}\right)^{f(x)} \approx \frac{1}{\mathrm{e}^{\frac{1}{c}} \cdot f(x)}.$$

The probability of any processor to become stable is clearly larger than P_{success}. From now on, we examine a set of Bernoulli trials, where one conducts several trials in parallel. The success probability in our case is the probability a processor has to become stable and the convergence time is the expected length of the longest trial. Since the probability of success is larger than P_{success}, the expected time for convergence is smaller than the expected length of the longest Bernoulli trial. As presented in [18], the expected value of the longest trial is $O(\log_{\frac{1}{(1-P_{\text{success}})}} n)$, where n is the number of concurrent trials held.

A note is in order regarding the dependability of processors. Since for every two processors within the same neighborhood the probabilities of success are dependent, using a reduction to Bernoulli trials only gives a correct upper bound since if one processor has succeeded, the probability of of a neighbor to succeed is increased.

To conclude, we can see that if $f(x)$ is constant, the expected convergence time of the synchronous algorithm is $O(\log n)$ phases. Since each phase is exactly x rounds, we get that the convergence time in terms of synchronous rounds is also $O(\log n)$. ■

4.2. *Asynchronous cluster construction*

We now present an asynchronous version of the previous hierarchy construction algorithm. Each processor p uses several key variables: $leader_p$, $candidate_p$, id_p and rtp_p. $leader_p$ denotes whether p is currently a leader. $candidate_p$ is set to true if p is trying to become a leader. id_p is the identifier each processor has, and rtp_p is a random temporary identifier used to break the symmetry between processors.

One may try using the processors' identifiers in order to break symmetry. However, occasionally an unfortunate order of id's may lead to a convergence time which is proportional to the diameter of the graph. We use randomness to break ties in order to overcome such a scenario.

The construction algorithm is composed of several parts. All processors participate in an (asynchronous) update algorithm up to distance x. Based on the update tables, each processor p constructs a tree rooted at p and of depth not exceeding x. Using the tree, each processor invokes the snapshot algorithm to collect the state of its neighborhood. We use the snapshot algorithm to perform a PIF algorithm, and by adding information to the markers used in the snapshot process we achieve the desired PIF effect. The number of trees and snapshot protocols each processor must participate in can be calculated from the topology collected earlier.

Constantly (this is to say that the time frame is not important), each processor p will take a snapshot of the surrounding neighborhood (up to distance x). After the snapshot is collected, the algorithm in Fig. 6 is invoked. Since the snapshot algorithm is guaranteed to be finished in each invocation (although the result might be incorrect, since the rooted tree has not stabilized yet), we are guaranteed that future invocations of the snapshot algorithm will take place. For a snapshot obtained at p, C_p, we denote $leader(C_p) = true$ if there exists a processor $q \neq p$ in C_p, such that $leader_q = true$.

Let us assume that a complete snapshot C_p is obtained at p. The four combinations of $leader_p$ and $leader(C_p)$ determine the course of actions p must follow. First, consider the most simple cases where $leader_p = false \wedge leader(C_p) = true$ or $leader_p = true \wedge leader(C_p) = false$. In these cases, p should avoid taking any action, since, as far as p can tell, the situation is correct. The complex cases are when there are no leaders in p's vicinity and p is not a leader itself or when p is a leader and can see another leader within a distance of x from itself. In case $leader_p = false \wedge leader(C_p) = false$, p will first choose a random number (from a predetermined range) and store it in rtp_p. Then, p will assign *true* to $candidate_p$ (Fig. 6 lines 3–4). The next operation is propagating the information that p wishes to become the leader of its neighborhood. This is achieved through the use of the snapshot protocol which results in a new snapshot at p, C'_p (line 5). Now, if C'_p does not contain information about a leader or another candidate, p can safely place itself as a leader and set $leader_p = true$. However, if $leader(C'_p) = true$ holds, p should set $candidate_p$ to *false*, since there is now a leader in p's neighborhood. Last, if there are other candidates in C'_p, p will become a leader if (and only if) the tuple $\langle rtp_p, id_p \rangle$ is larger than all other candidate's tuples in C'_p (line 10).

The last case is when $leader_p = true \wedge leader(C_p) = true$ (line 16). Upon detecting such a condition, p will immediately assign $leader_p$ and $candidate_p$ with *false* and will start a new cycle of the algorithm.

To prove that the asynchronous hierarchical construction algorithm works, we will take an approach similar to the proof of the synchronous algorithm. We will denote a processor p as *stable* in two cases. The first case is a stable leader, when $leader_p = true, \forall q \in neighborhood_p(leader_q = false)$. Furthermore, all topology tables for each processor within p's neighborhood are up to date and reflect p's leadership and no other message exists in the system denoting another

```
Predicates:
leader(C_p) :=
    ∃q ∈ C_p | q ≠ p ∧ leader(q)

1 (leader_p ⊕ leader(C_p)) = true:
    /* do nothing (stable). */

2 leader_p = false ∧ leader(C_p) = false:
3   rtp_p ← random()
4   candidate_p ← true
5   C'_p ← new snapshot
6   if leader(C'_p) = true then
7   |   candidate_p ← false
8   |   leader_p ← false
9   else if ∀q ∈ C'_p candidate_q = true →
10  |   (⟨rtp_q, id_q⟩ < ⟨rtp_p, id_p⟩) then
11  |   leader_p ← true
12  else
13  |   candidate_p ← false
14  |   leader_p ← false
15  end

16 (leader_p = true ∧ leader(C_p) = true):
17   candidate_p ← false
18   leader_p ← false
```

Fig. 6. Asynchronous leader election algorithm for processor p.

processor as leader or candidate. The second case is of a stable node, when p is not a leader, but there is a stable leader in p's neighborhood.

Our first claim is that once a processor is stable, it will remain stable. Next, we show that a processor has a positive probability of becoming stable. We then use the scheduler-luck game to show that the algorithm stabilizes ([9], chapter 2.9)

Lemma 9. *Let $\mathcal{E} = (c_0, a_0, c_1, a_1, \ldots)$ be a fair execution. If at a global configuration c_i, a processor p has become stable, then p will remain stable for all configurations $c_j \in E$, such that $j > i$.*

Proof. First, let us consider the case in which p is a stable leader in c_i. From the definition of a stable leader, for each $q \in neighborhood_p$, $leader_q = false$ and q denotes p as a leader in q's update tables. Now, since the topology tables do not change, p can lose the stability property only if another processor within p's neighborhood becomes a leader too. Assume that such a processor q becomes a leader in $c_j, j > i$. Based on the leader election algorithm, this is possible in two cases. Either q was not a leader and did not see a leader in its neighborhood, or q was a leader and saw another leader. The first option is not feasible, since q is aware of p being a leader (from the definition of stable). The second option is not possible either, since q will not set itself a leader as long as p is. Hence, a stable leader will remain stable.

Now, we will proceed to discussing a stable node p (not a leader). From the definition of a stable processor we deduce that there is a stable leader q in the neighborhood of p. Since q is a stable leader and will remain such, p will remain stable too. ■

Lemma 10. *Let $\mathcal{E} = (c_i, a_i, c_{i+1}, a_{i+1}, \ldots)$ be a fair execution, such that the update algorithm and the snapshot algorithm have stabilized. Starting from any configuration in $\mathcal{E}$, each processor p has a positive probability to become stable within $O(x)$ rounds.*

Proof. The stabilization time for the topology update algorithm and the snapshot algorithm is $O(x)$. Consider now a processor p which is not stable in c_0. Let c_i be the first configuration in which p has obtained a correct snapshot of its neighborhood, C_p (obviously, c_i is reached within $O(x)$ rounds). If p is stable in c_i, then the process has been completed. Otherwise, we will show that p has a positive probability of becoming stable within $O(x)$ rounds.

We will now show that within $3 \cdot x$ rounds at most, either p becomes stable or there exists a processor in p's neighborhood which chooses a new *rtp* value and has a positive probability of becoming a stable leader. Assume, towards contradiction, that no processor in p's neighborhood chooses a new *rtp* value within $3 \cdot x$ rounds and p does not become stable during this time. If there were no leaders in p's neighborhood, p would have chosen a new *rtp* value within x rounds (the number of rounds which would take p to finish a new snapshot). Hence, there exists a processor $q \in neighborhood_p$ such that $leader_q = true$ (it is possible that $q = p$). Since p is not stable, we can deduce that q is also not a stable leader. Suppose, that after x rounds, q has not set $leader_q$ to *false* and q is still not stable. This implies that a different processor which is also a leader exists in q's neighborhood. After x rounds at most, q will detect this fact by way of a new snapshot and will set $leader_q$ to *false*. In a similar manner, we can show that each leader in p's neighborhood eventually either becomes stable within $2 \cdot x$ rounds or relinquishes leadership. If one leader becomes stable, the proof is now completed. Otherwise, a processor in p's

neighborhood (possibly p) will notice the fact that there are no leaders and will choose a new *rtp* value within x rounds. Overall, we get that after $O(x)$ rounds at least one new *rtp* value is chosen.

Denote p_r as the processor which first chooses a new *rtp* value in p's neighborhood. This assignment is a result of p_r finishing a snapshot C_r (which takes $O(x)$ rounds to complete) and of noticing that no leaders exist in this snapshot. Our next claim is that between the start of the snapshot that resulted in C_r and the end of the next snapshot that p_r will take (line 5) and which is denoted C'_r, each processor in p_r's neighborhood cannot assign more than one new value to its *rtp* if the right conditions hold. Denote c_{start} as the configuration in which the snapshot C_r started and c_{end} as the configuration in which C'_r ended. Assume that each processor q, which chooses a new *rtp* value between c_{start} and c_{end} chooses a value smaller than that of p_r. Once the PIF snapshots initiated by p_r reach q, q loses to p_r, and will not enter line 2. Thus, q will not choose a new random *rtp* value more than once between c_{start} and c_{end}. The probability that p_r will choose an *rtp* value in such a way is larger than P_{success}. As a result, p_r will assign $leader_{p_r}$ by *true* (line 10) and within x rounds will become a stable leader. This way, p will also become stable within $4 \cdot x$ rounds from c_0. ■

Thus, we can make the following corollary:

Corollary 1. *In every fair execution, each processor has a positive probability of becoming stable in every $O(x)$ rounds and it holds by [18] that within $O(\log n)$ expected number of rounds the algorithm converges to a stable state.*

4.3. Hierarchy construction

Constructing the hierarchy is achieved by a repeated application of the clustering algorithm. We suggest using the clustering algorithm on the original graph G, constructing clusters with $x > 1$ (in essence, a minimal x-dominating set). We then propose to dynamically define an overlay network between the leaders of each cluster and apply the same scheme to the resulting graph. The process is completed after a single cluster, composed of the entire graph G, is finally defined. The resulting hierarchy is of $O(\log n)$ levels, and in each level i (level 0 is the original graph, G) there exist at most $\frac{n}{2^i}$ processors. This bound arises from the fact that each leader p has at least one processor directly connected to p, which is not directly connected to any other leader. Since there exist $O(\log n)$ levels in the hierarchy and since communication on overlay edges is considered non expensive, the hierarchy construction algorithm stabilizes within $O(\log^2 n)$ expected rounds ($O(\log n)$ for each level, times $O(\log n)$ levels), assuming the degree of each of the hierarchy levels is bounded.

Next, we describe the construction of the overlay network and present a graph class in which the degree of each hierarchy level is bounded.

4.3.1. Overlay network construction

Let $G = G_0 = (V_0, E_0)$ be the original graph, to which we apply our clustering algorithm. We define $G_i = (V_i, E_i)$ so that $V_i = \{p \in V_0 | \, p \text{ is a leader in } V_{i-1}\}$ and $(p, q) \in E_i$ iff, the length of the shortest path between p and q in G_0 is at most $2 \cdot x^i + x^{i-1}$ (where x is the parameter of the clustering algorithm). This construction can be easily achieved by each leader p by extending the update algorithm to include processors up to distance $x + 1$ (instead of x) and adding the list of leaders at distance x to each processor p to p's tuple. We then apply the clustering algorithm on G_i, so that leaders will dominate processors up to distance x^{i+1} in G_0. Note that the criteria for distance among leaders is expressed in terms of G_0 and the original x, namely; x^{i+1} for level i of the hierarchy.

Lemma 11. *Each resulting graph G_i is a connected graph.*

Proof. By induction: G_0 is a connected graph, by definition. Assume G_{i-1} is also a connected graph, and G_i is the result of the clustering algorithm. Let p_0 and p_k be processors in V_i such that $p_0, p_1, \dots, p_k$ is a path between p_0 and p_k in G_{i-1} (such a path exists, since G_{i-1} is a connected graph). Let q_j be the chosen leader of p_j $(1 \le j \le k-1)$ in G_{i-1}. According to the overlay construction, $(p_0, q_1) \in E_i \wedge (q_{k-1}, p_k) \in E_i$. Furthermore, $\forall 2 \le j \le k-1$ $(q_{j-1}, q_j) \in E_i$, since the distance in G_0 between p_{j-1} and p_j is at most $2 \cdot x^2 + x^{i-1}$ (or $p_{j-1} = p_j$). Hence, $p_0, q_1, q_2, \dots, q_{k-1}, p_k$ is a path between p_0 and p_k in G_i. ■

To obtain higher levels of the hierarchy, we continue with the same construction recursively. Suppose we have defined the levels of the hierarchy up to (and including) level i. The processors of G_i will participate in the clustering algorithm up to distance x^{i+1}. G_{i+1} will be composed of the resulting leaders of G_i, such that two processor are neighbors iff, the length of the shortest path between them in G_0 is at most $2 \cdot x^{i+1} + x^i$. Each G_i is, in turn, also connected, according to Lemma 11. To realize this construction, we suggest each leader p will add to its update table of G_i all the topology p has collected in each G_j up to now.

Next, we describe the *geographically affined* class of graphs such that the clustering algorithm and the overlay construction, applied on these graphs, produces an overlay graph of bounded degree. This class is implied by a typical deployment of sensor networks.

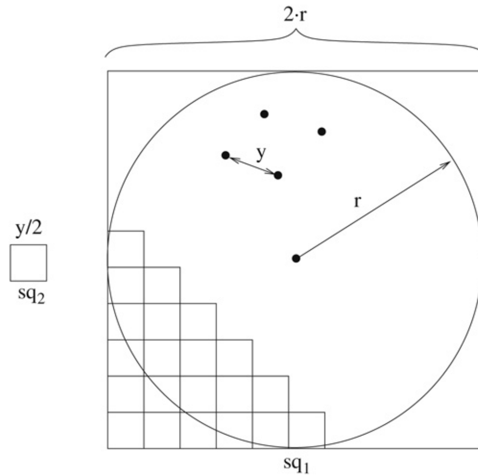

Fig. 7. Maximal number of leaders.

4.3.2. Geographically affined graphs

In this class of graphs we wish to explore the relation between the Euclidean distance between processors and the length of the shortest path between them. This definition is similar to the embedding schemes presented in [23]. We first define the *geographically affined* class of graphs.

Definition 4.1. Let $G = (V, E)$ be a graph embedded in the Euclidean plane. For $p, q \in V$, define $\|(p, q)\|_2$ as the Euclidean distance between p and q, and $dist(p, q)$ as the number of hops in a shortest path from p to q in G. G is *Geographically affined* iff, there exist a constant $c \leq 1$ such that $\forall p, q \in V : c \cdot dist(p, q) \leq \|(p, q)\|_2 \leq dist(p, q)$.

We will next show that each geographically affined graph has a bounded degree. Furthermore, we also show that the hierarchy construction algorithm presented above produces a bounded degree graph in each level of the hierarchy.

Lemma 12. *Given a circle C of radius r and a set S of points in C, where the minimal distance between any two points is y, then $|S| \leq \frac{16 \cdot r^2}{y^2}$.*

Proof. Consider Fig. 7. C is contained in a square sq_1 whose edges are of length $2 \cdot r$. In each square sq_2 whose edges are of length $\frac{y}{2}$, there can be at most one point from S. $|S|$ is obviously smaller than the number of sq_2 squares which can be fitted into sq_1. Hence, $|S| \leq \frac{4 \cdot r^2}{y^2/4} = \frac{16 \cdot r^2}{y^2}$. ■

Lemma 13. *Let $G_0 = (V_0, E_0)$ be an Euclidean graph, such that G_0 is geographically affined. Each graph in the series $\{G_i\}_{i=0}^{\log n}$, resulting from the consecutive application of the clustering algorithm with parameter x^{i+1}, has a degree at most $\frac{16}{c^2} \cdot (2 \cdot x + 1)^2$.*

Proof. Let p be a processor in G_i, and N_p the set of p's neighbors in G_i. The shortest path (in G_0) between p and a neighbor q is at least x^i (since in G_{i-1} p and q are leaders) and at most $2 \cdot x^{i+1} + x^i$ (p and q are neighbors in G_i iff, their distance in G_{i-1} is at most $2 \cdot x + 1$ hops, which is at most $2 \cdot x^{i+1} + x^i$ hops in G_0). In a similar fashion, the shortest path (in G_0) between any $q, r \in N_p$ is at least x^i (if they are neighbors in G_i). Since the graph is geographically affined, we get the following equations:

$$c \cdot x^i \leq \|(p, q)\|_2 \leq 2 \cdot x^{i+1} + x^i$$
$$c \cdot x^i \leq \|(q, r)\|_2.$$

Hence, each $q \in N_p$ must reside inside a circle C, centered at p and of radius $2 \cdot x^{i+1} + x^i$. According to Lemma 12, $|N_p|$ is bounded by $16 \cdot \frac{(2 \cdot x^{i+1} + x^i)^2}{(c \cdot x^i)^2} = \frac{16}{c^2} \cdot (2 \cdot x + 1)^2$. ■

4.4. Self-organization properties

Next, we prove that our algorithms are self-organizing. Firstly, for the clustering algorithm, it is worthwhile noting that locality holds since the algorithm stabilizes within expected $O(\log n)$ rounds. Thus, we focus our discussion on *dynamic* changes of the communication graph — namely, on addition and removal of communication links. We wish to draw the readers' attention to the fact that addition (or removal) of processors can be modeled by the addition (or removal) of their communication links (which is a bounded number of operations). When we discuss addition of processors, we consider addition of processors in a predefined state or in an arbitrary state. We only consider topology changes after the algorithm has stabilized (otherwise, the global stabilization time applies).

Lemma 14. *Starting in a safe configuration of the clustering algorithm, if the update table of processor p has changed due to a channel (respectively, processor) addition or removal in configuration c_i and the channel (respectively, processor) is attached (a neighbor) to p, then within expected $O(x + \log f(x)) = O(1)$ rounds, a safe configuration is reached. Furthermore, for each processor q, such that $dist(p, q) > 2 \cdot x$, q will remain stable.*

Proof. Let us assume that p is a processor as described in the Lemma. Since the update tables of each processor are restricted to processors of distance x, no processor q, such that $dist(p, q) > 2 \cdot x$, will change its own table. Furthermore, after $O(x)$ rounds, each processor will have correct tables. The clustering algorithm now has to stabilize only in the small neighborhood of p, which takes $O(\log f(x))$ expected number of rounds.

Next, let us assume that q was a stable leader in c_i and $dist(p, q) > 2 \cdot x$. First, we argue that each processor r in q's neighborhood, maintains a correct tuple of the update algorithm denoting q. This is obvious, since no topology changes were made in q's neighborhood. As a result, q will correctly participate in each spanning tree constructed by such a processor r. Consequently, each time r takes a snapshot of its neighborhood, r will see that $leader_q = true$. Hence, r will not assign *true* to $leader_r$ and q will remain a stable leader. Thus, all processor within q's neighborhood will remain stable. ■

We now consider the effects that channel additions have on the clustering algorithm. Let us assume that a new (bi-directional) channel, (p, q), is added between processors p and q. We argue that any stable processor distanced more than $2 \cdot x$ from either p or q will remain stable. Furthermore, within an expected constant number of rounds, the algorithm will stabilize. This clearly follows from Lemma 14. Let us now assume that a channel (p, q) is removed. Let NL be the set of all processors, so that the removal of (p, q) leaves them leaderless or unstable. We argue that the constant number of processor in NL are at most at distance x from either p or q and that stable processors which are distanced farther than x will remain stable. Processor removal is easily reduced to the removal of all channels attached to this processor from the communication graph.

We also discuss additions and removals of processors. We argue that stable processors which are farther than $2 \cdot x$ from the removed/added processor will remain stable. This also clearly follows from Lemma 14.

Thus, our clustering algorithm is self-organizing, since the expected convergence time is $O(\log n) \in o(n)$ and the number of processors which change state due to a dynamic topology change is constant. In fact, when k changes occur approximately at the same time, the expected convergence time is $O(\log k)$ following the last change occurrence.

Application to hierarchy; Let us examine a dynamic change at G_0. There are two processors, p and q, which are involved in the change ((p, q) was either added or removed). We first concentrate on p. From Lemma 14 we infer that only processors within a distance of $2 \cdot x + 1$ hops from p can be affected in G_0. The dynamic change can influence the state of leaders within this range, which can be regarded as a new dynamic change in G_1. The radius of the corresponding influenced region from p in G_1 is therefore $(2 \cdot x^2 + 2 \cdot x + 1) + (2 \cdot x + 1)$ around p in G_0. In a similar way, the radius of the influenced region from p in G_i is $2 \cdot x^i + 2 \cdot x^{i-1} + x^{i-2} +$ (the radius of influence in G_{i-1}). Overall, the area of effect around p in G_0 is less than $4 \cdot x^{i+2}$. Since G_0 is geographically affined, the Euclidean radius of such a circle is smaller than $4 \cdot x^{i+2}$. The minimal distance in G_0 between processor in G_i is at least x^i (when counting real edges, not virtual ones), since they are leaders in G_{i-1}. Again, since G_0 is geographically affined, the Euclidean distance between leaders is at least $c \cdot x^i$. Using Lemma 12, it is evident that the number of processors affected at G_i because of p is at most $\frac{16 \cdot (4 \cdot x^{i+2})^2}{(x^i)^2} = 256 \cdot x^4 = O(1)$. Since we have to consider q as well, we double the total number of changes to have a total of $O(1)$ changes in each level.

To conclude, the hierarchy construction algorithm is self-organizing, since the expected stabilization time is $O(\log^2 n) \in o(n)$ and dynamic topology changes affect only $O(\log n) \in o(\log^2 n)$ processors. Similarly, when k changes occur approximately at the same time, the expected convergence time is $O(\log^2 k)$ rounds following the last occurring change.

5. Overlay based hierarchical snapshot algorithm

We now present a self-stabilizing and self-organizing snapshot scheme (which also enables subsystems to take snapshots independently). Due to the use of overlay links, the resulting snapshot is sublinear.

Let p be a node in $\mathcal{HT}$, so that p is a parent of leaves in $\mathcal{HT}$. Let $p_1, p_2, \ldots, p_k$ be the children of p in $\mathcal{HT}$. Note that p and $p_1, p_2, \ldots, p_k$ reside in the same subsystem, $subs_i$, which is a connected component of G. A spanning tree of $subs_i$ rooted at p is constructed and p is responsible for invoking snapshots in $subs_i$.

Let q be a node in $\mathcal{HT}$, so that at least one child of q in $\mathcal{HT}$ is a subsystem, consisting of more than a single processor. Let us assume that q represents (is the leader of) the subsystem *subs*. Let $subs_1, subs_2, \ldots, subs_j$ be the subsystems represented by the children of q in $\mathcal{HT}$. Note that the union of $subs_1, subs_2, \ldots, subs_j$ is identical to *subs*, the subsystem represented by q. Let $q_1, q_2, \ldots, q_l$ be the processors that are leaders of $subs_1, subs_2, \ldots, subs_l$, respectively. It is important to note that it is possible that the processor q, that is the leader of *subs*, may also serve as a leader q_i of (at most) one of the above $subs_i$. Using the communication links in *subs*, we define an overlay network connecting $q, q_1, q_2, \ldots, q_l$. A spanning tree, rooted at q, of the obtained overlay network is constructed. q is responsible for invoking snapshots in *subs* using the spanning tree of the overlay network of *subs*. When a snapshot is requested at q, it will initiate a snapshot in *subs*.

The snapshot initiated at *subs* can serve two purposes. On one hand, the snapshot algorithm of *subs* can be used to obtain a consistent snapshot *only* of $q_1, q_2, \ldots, q_l$ and q. On the other hand, when q sends a marker, it can add another indicator

bit, which acts as a snapshot request for $q_1, q_2, \ldots, q_l$ in $subs_1, subs_2, \ldots, subs_l$. The result is a recursive invocation of the snapshot algorithm, resulting in a consistent snapshot of all processors in *subs*.

The overlay network inside subsystem *subs* is needed to ensure fifo delivery of messages between leaders. We also need to restrict cross subsystem communications (of the distributed algorithm) to travel *only* through the subsystem leaders using the overlay network. This is done in order to ensure that messages will not be able to bypass markers or to corrupt snapshots.

The addition of the overlay network requires several adjustments of the snapshot algorithm. The overlay network adds *virtual links* to the communication graph which should be also recorded. In order to implement virtual links, routing information such as the one used in the source routing scheme, must be added to messages. Consequently, each processor, upon receiving a message through a physical link, can decide which virtual link this message belongs to. Furthermore, the state of the processor is not affected by the arrival of this message, since it is only forwarded to its destination. Hence, the processor can ignore this message with regards to the snapshot algorithm without recording it on the physical link. Such messages need to be recorded only at their destination, on the virtual link they traverse on.

6. Extensions and concluding remarks

Self-organization. We have given a simple and intuitive definition of self-organization. Furthermore, we have displayed the relevance of self-stabilization with regard to self-organization. Our self-stabilizing and self-organizing snapshot algorithm implies sublinear time algorithms in the overlay network model for many core distributed tasks.

Self-stabilizing and self-organizing leader election. The hierarchy construction algorithm which is, by itself, a self-stabilizing and self-organizing algorithm, naturally defines a leader for each subsystem. Thus, the topmost subsystem (which contains the entire system) also has a leader, which we define to be the output of the leader election algorithm. Hence, the output of the hierarchy construction algorithm can be used to define a self-stabilizing leader election algorithm which converges in $O(\log^2 n)$ expected number of rounds and handles topology changes gracefully in $O(\log n)$ rounds.

Our definition of self-organization can easily capture the effect of transient faults on the system. It can be shown that a single transient fault in the system can effect only the local updates of a constant number of processors and therefore influence $O(1)$ states. Moreover, the number of state changes following (approximately) simultaneous faults that occur in neighboring processors is proportional to the group's diameter in the graph. In the worst case, when the faults are approximately x apart (say, all leaders change state to non-leaders) the number of faults is $O(n)$ allowing a complete stabilization phase.

Self-stabilizing and self-organizing snapshots. Building on top of the hierarchy construction algorithm, we have presented a self-stabilizing snapshot scheme, where a global snapshot can be collected in $O(\log^2 n)$ rounds (in fact, if the hierarchy was previously defined, only $O(\log n)$ rounds are necessary).

Self-stabilizing converter. Our self-stabilizing and self-organizing snapshot algorithm implies a new efficient tool for converting distributed (reactive, or fixed output) algorithms to self-stabilizing algorithms in sublinear time; the leader of the system can take repeated snapshots and verify each snapshot for correctness. When a snapshot indicates an illegal state, a global reset procedure may be initiated, using the infrastructure created by the hierarchy definition algorithm, to reach a predefined (and safe) state.

Acknowledgments

Many thanks to Noga Alon for helpful discussions. The first author was partially supported by IBM, Israeli ministry of science, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences. The second author was partially supported by Deutsche Telekom, Israeli Grid Consortium and the Lynne and William Frankel Center for Computer Sciences.

References

[1] Y. Afek, S. Dolev, Local stabilizer, in: Self-stabilizing Distributed Systems, Journal of Parallel and Distributed Computing 62 (5) (2002) 745–765 (special issue). Also in Proc. of the 5th Israeli Symposium on Theory of Computing and Systems, ISTCS 1997, 1997, pp. 74–84.
[2] E. Anceaume, X. Defago, M. Gradinariu, M. Roy, Towards a theory of self-organization, in: 9th International Conference on Principels of Distributed Systems, OPODIS, 2005, pp. 146–156.
[3] J. Burman, S. Kutten, T. Herman, B. Patt-Shamir, Asynchronous and fully self-stabilizing time-adaptive majority consensus, in: 9th International Conference on Principels of Distributed Systems, OPODIS, 2005.
[4] M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Transactions on Computing Systems 3 (1) (1985) 63–75.
[5] A. Tanenbaum, Computer Networking, 4th ed., Prentice Hall, 2002.
[6] A. Cournier, A. Datta, F. Petit, V Villain, Enabling snap-stabilization, in: Proc. of the 23rd International Conference on Distributed Computing Systems, 2003, pp. 12–19.
[7] A. Costello, G. Varghese, Self-stabilization by window washing, in: Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing, 1996, pp. 35–44.
[8] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Communications of the ACM 17 (11) (1974) 643–644.
[9] S. Dolev, Self-Stabilization, MIT Press, 2000.
[10] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, in: Proc. of the 2nd Workshop on Self-Stabilizing Systems, May 1995, Chicago Journal of Theoretical Computer Science 3 (4) (1997) special issue on Self-stabilization.

[11] S. Dolev, A. Israeli, S. Moran, Resource bounds for self-stabilizing message driven protocols, in: Symposium on Principles of Distributed Computing, 1991, pp. 281–293.
[12] S. Dolev, E. Kranakis, D. Krizanc, D. Peleg, Bubbles: Adaptive routing scheme for high-speed dynamic networks, SIAM Journal on Computing 29 (3) (1999) 804–833. Also in Proc. of the 27th ACM Symposium on Theory of Computing, STOC 1995, 1995, pp. 528–537.
[13] Felix C. Gärtner, Henning Pagnia, Time-efficient self-stabilizing algorithms through hierarchical structures, in: Proc. to the Sixth Symposium on Self-Stabilizing Systems, 2003, pp. 154–168.
[14] S. Ghosh, A. Gupta, T. Herman, S. Pemmaraju, Fault-containing self-stabilizing algorithms, in: PODC, 1996, pp. 45–54.
[15] A. Goldberg, S. Plotkin, G. Shannon, Parallel symmetry- breaking in sparse graphs, in: Proceedings of the Nineteenth Annual ACM Conference on theory of Computing, STOC, 1987, pp. 315–324.
[16] M. Henzinger, V. King, Randomized fully dynamic graph algorithms with polylogrithmic time per operation, Journal of the ACM 46 (4) (1999) 502–516.
[17] S. Katz, K. Perry, Self-stabilizing extensions for message-passing systems, in: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, 1990, pp. 91–101.
[18] P. Kirschenhofer, H. Prodinger, A result in order statistics related to probabilistic counting, in: Computing, vol. 46 pp. 15–27.
[19] S. Kutten, D. Peleg, Tight fault locality, in: Annual Symposium on Foundations of Computer Science, FOCS, 1995.
[20] F. Kuhn, T. Moscibroda, T. Nieberg, R. Wattenhofer, Fast deterministic distributed maximal independent set computation on growth-bounded graphs, in: Distributed Computing: 19th International Conference, DISC 2005, pp. 273–283.
[21] F. Kuhn, T. Moscibroda, R. Wattenhofer, What cannot be computed locally!, in: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, PODC, 2004, pp. 300–309.
[22] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 12 (7) (1978) 558–565.
[23] N. Linial, E. London, Y. Rabinovich, The geometry of graphs and some of its algorithmic applications, in: Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science, October 1994, pp. 577–591.
[24] M. Luby, A simple parallel algorithm for the maximal independent set proble, SIAM journal of Computing 15 (4) (1986) 1036–1053.
[25] T. Moscibroda, R. Wattenhofer, Efficient computation of maximal independent sets in unstructured multi-hop radio networks, in: The 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems, Fort Lauderdale, Florida, 2004.
[26] C.G. Plaxton, R. Rajaraman, A.W. Richa, Accessing nearby copies of replicated objects in a distributed environment, in: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1997, pp. 311–320.
[27] G. Varghese, Self-stabilization by counter flushing, SIAM Journal on Computing 30 (2) (2000) 486–510. Also in: Symposium on Principles of Distributed Computing, 1994, pp. 244–253.
[28] H. Zhang, A. Arora, GS^3: Scalable self-configuration and self-healing in wireless networks, in: Symposium on Principles of Distributed Computing, 2002, pp. 58–67.