EMPIRE OF COLONIES
Self-Stabilizing and Self-Organizing Distributed Algorithms
(Extended Abstract)

Shlomi Dolev and Nir Tzachar
Department of Computer Science, Ben-Gurion University of the Negev, Israel.
dolev@cs.bgu.ac.il, tzachar@cs.bgu.ac.il

Abstract. 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 sublinear 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 snapshot. 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 combination of the self-stabilizing and self-organizing distributed hierarchy construction and the snapshot algorithm form 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.

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

 Partially supported by IBM, Israeli ministry of science, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Israeli Grid Consortium and the Lynn and William Frankel Center for Computer Sciences.
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 automatically recover from an inconsistent state, namely to be self-stabilizing ([11]) 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 [12]) 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 regards 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 \( d(n) \in o(s(n)) \).

In this work, we enable algorithms to define (on the fly) and then 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, as (almost) no processing is involved in forwarding messages over these links (e.g., [13, 26], MPLS [6]).

**Main 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.

- **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 bigger 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., [7]) 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 networks 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

\(^1\) In some cases, preassigned frequencies or/and supporting switching hardware can be used. e.g., MPLS–[6].
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.

**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., [20, 16, 1, 4]). 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. A constant time clustering algorithm is presented in [8]. The algorithm assumes that processors can measure time and therefore does not fit asynchronous systems.
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.

Applications of hierarchy in the self-stabilization domain are described in [15]. 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.

- **Snap-stabilization.** Snap-stabilizing algorithms were first introduced in [3]. A protocol is said to be snap-stabilizing if, upon the first invocation of the protocol by one processor, the protocol behaves according to its specifications. The snapshot algorithm we present is snap-stabilizing, provided specific preconditions are met: Namely, a tree structure is defined and a leader is present beforehand. When the leader invokes a snapshot, the snapshot terminates with a correct answer.

- **Dynamic graph algorithms.** Extensive research on distributed dynamic algorithms appeared in the literature (e.g., [13] 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., [17] and the reference therein). We achieve a committing time (logarithmic and polylogarithmic) in (fault tolerance) distributed settings for an important class of graphs.

**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. Conclusions appear in Section 5. More details and most of the proofs are omitted from this extended abstract and can be found in [14].

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 \( \text{dist} \) of two processors \( p \) and \( q \), \( \text{dist}(p, q) \), 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 \( \text{dist}(p, q) \leq x \). 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.

Overlay edges. 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 almost no processing is required by intermediate processors in order to forward the message 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 message to traverse such an overlay link as the time for traversing a link that directly connects two neighboring processors. The 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. In such a case, messages are buffered only at the endpoints, resulting in an overly link of the same capacity as the original links.

We assume 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 \( m \).

A configuration \( c \) of the system is a tuple \( c = (S, L) \); \( S \) is a vector of states, \( \langle s_1, s_2, \cdots, s_n \rangle \), where the state \( s_i \) is a state of processor \( p_i \); \( L \) is a vector of link states \( \langle l_{i,j}, \cdots \rangle \) for each \( (i, j) \in E \). 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, \( E = \{c_0, a_0, c_1, a_1, \cdots \} \), 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 \( E \) is fair if every processor executes a step infinitely often in \( E \) 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 with 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.

The snapshot task \( S \) for a system is defined by a set of executions \( E_S \) 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 $E'$ that starts immediately before $a_r$, reaches $gs$ and then continues to the configuration $c_s$. $E'$ may be different from the execution which actually took place.

We use the notion of asynchronous rounds to measure the time complexity of an algorithm. The first asynchronous round in execution $E$ is the shortest prefix of $E$ 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 $E$ is the first asynchronous round of the suffix of $E$ that immediately follows the first asynchronous round in $E$. 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

Our starting point is the unbounded snapshot algorithm presented in [18] and the snap-stabilizing algorithm presented in [7] 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 [18] 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 snapshot algorithm uses, as a building block, a snap-stabilizing data link algorithm which is specifically designed for bounded capacity links (the data link algorithm appears in [14]. The algorithm uses three variables to control the data flow on the link. $current[q]$ holds the current value which is sent to $q$, $next[q]$ holds the next value to be sent, which is suspended until an acknowledgment on $current[q]$ arrives. $last[q]$ holds the last acknowledged message. The snapshot algorithm then uses $next[q]$ to send messages to a neighbor $q$, and waits for an acknowledgment in $last[q]$. To ensure self-stabilization, a sequence number is attached to each message sent, and is incremented by one modulo two times the link capacity plus one for each new message. To ensure snap-stabilization, each message is sent repeatedly two times the link capacity plus one – a step that ensures that the message had arrived and that the acknowledgment is valid; if $lc$ is the link capacity, then there can be at most $2 \cdot lc$ messages in transit on the link. By using $2 \cdot lc + 1$ labels, one label is guaranteed to be a new label, which does not exist in the link. This, in turn, ensures that a correct acknowledgment is received.
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 [5]. 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). 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 [5].

A detailed description of the algorithm, as well as the correctness proof and complexity analysis, appears in [14]. We mention here the main properties of the snapshot algorithm:

**Theorem 1.** Once the root of the tree initiates a snapshot cycle, a correct snapshot will be obtained in $O(h)$ rounds, where $h$ is the height of the tree.

### 4 Hierarchical Construction Schemes

A hierarchical system is represented by a communication graph, $G = (V, E)$ and a hierarchy tree $HT = (V_h, E_h)$. Each node in $HT$, $l_i$, represents a set of nodes in $V$, called a subsystem, so that if $l_i$ and $l_j$ are at the same level of $HT$, then $l_i \cap l_j = \emptyset$. Furthermore, if $K$ is a set of nodes at level $i$ of $HT$, then $\bigcup_{j \in K} l_j = V$. 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 size smaller than a fixed parameter. Furthermore, each cluster is defined by a “native” leader.

- **Cluster construction.** The clustering algorithm we present is a maximal independent set algorithm, where each dominator is denoted as a leader and each dominatee joins the closest dominator (ties are broken by, say, leaders’ identifiers).
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 $x$. 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 $dist = k \leq x$, such that there exist entries with $dist = j$ for all $j < k$. All other entries are deleted. The removal of entries ensures fast stabilization [12], as an entry which is $j$ hops away from $p$ must be connected to $p$ by a path, so there must exists entries which are $1, 2, \ldots, j - 1$ hops from

```
Predicates:
leader(C_p) :=
\exists q \in C_p | q \neq p \land leader(q)
1 (leader_p \oplus leader(C_p)) = true:
/* do nothing (stable). */
2 leader_p = false \land 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 \land
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 \land leader(C_p) = true):
17 candidate_p ← false
18 leader_p ← false
```

Fig. 1: Asynchronous Leader Election Algorithm for Processor $p$
p. Afterwards, p adds 1 to the distance field of every tuple and finally adds the tuple \( \langle id_p, 0, \text{nil} \rangle \) 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, \( \text{leader}_p, \text{rtp}_p \). 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 \( \text{dist} > x \). 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.

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 Figure 1 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 \( \text{leader}(C_p) = \text{true} \) if there exists a processor \( q \neq p \) in \( C_p \), such that \( \text{leader}_q = \text{true} \).

Let us assume that a complete snapshot \( C_p \) is obtained at \( p \). The four combinations of \( \text{leader}_p \) and \( \text{leader}(C_p) \) determine the course of actions \( p \) must follow. First, consider the most simple cases where \( \text{leader}_p = \text{false} \land \text{leader}(C_p) = \text{true} \) or \( \text{leader}_p = \text{true} \land \text{leader}(C_p) = \text{false} \). 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 \( \text{leader}_p = \text{false} \land \text{leader}(C_p) = \text{false} \), \( p \) will first choose a random number (from a predetermined range) and store it in \( \text{rtp}_p \). Then, \( p \) will assign \( \text{true} \) to \( \text{candidate}_p \) (Figure 1 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); such propagation can be achieved by piggy-backing information on the markers of the snapshot. Now, if \( C'_p \) does not contain infor-
mation about a leader or another candidate, \( p \) can safely place itself as a leader and set \( \text{leader}_p = \text{true} \). However, if \( \text{leader}(C'_p) = \text{true} \) holds, \( p \) should set \( \text{candidate}_p \) to \( \text{false} \), 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 \( \text{leader}_p = \text{true} \wedge \text{leader}(C_p) = \text{true} \) (line 16). Upon detecting such a condition, \( p \) will immediately assign \( \text{leader}_p \) and \( \text{candidate}_p \) with \( \text{false} \) and will start a new cycle of the algorithm.

The correctness proof, as well as the time complexity analysis, appears in [14]. We only mention 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 [19] that within \( O(\log n) \) expected number of rounds, the algorithm converges to a stable state.

- **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.

- **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 \( V_i = \{ p \in V_0 \mid p \text{ is a leader in } V_{i-1} \} \) 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 1.** Each resulting graph $G_i$ is a connected graph.

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.

- **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 41.** 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 $\text{dist}(p, q)$ 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 $\forall p, q \in V : c \cdot \text{dist}(p, q) \leq \|(p, q)\|_2 \leq \text{dist}(p, q)$.

In [14], we 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. The proof of the next Lemma appears in [14].

**Lemma 2.** 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 $\frac{16}{c^2} \cdot (2 \cdot x + 1)^2$.

**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 3. 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 $\text{dist}(p, q) > 2 \cdot x$, $q$ 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 3. 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 3.

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 3 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 $2 \cdot x^i + 2 \cdot x^{i-1} + x^{i-2} + \ldots + (2 \cdot x + 1)$ (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 simple geometric arguments (see [14]) 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 Conclusions

We have given a simple and intuitive definition of self-organization. Furthermore, we have displayed the relevance of self-stabilization with regards 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.

Self-stabilizing and self-organizing snapshots. Building on top of the hierarchy construction algorithm, we have presented in [14] 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 and Boaz Patt-Shamir for helpful discussions.
References


