Time-Efficient Self-Stabilizing Algorithms through Hierarchical Structures

Felix C. Gärtner[1]* and Henning Pagnia[2]

[1] Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Distributed Programming Laboratory, CH-1015 Lausanne, Switzerland, fcg@acm.org

[2] University of Cooperative Education, D-68163 Mannheim, Germany, pagnia@computer.org

Abstract. We present a method of combining a self-stabilizing algorithm with a hierarchical structure to construct a self-stabilizing algorithm with improved stabilization time complexity and fault-containment features. As a case study, a self-stabilizing spanning-tree algorithm is presented which in favorable settings has logarithmic stabilization time complexity.

1 Introduction

A common prejudice of the self-stabilization concept is that the notion is too demanding to be of practical use. Self-stabilizing systems are required to recover to a set of legal states starting from *any* initial configuration and in *every* execution [4]. Consequently, weaker notions of self-stabilization have been devised such as *pseudo-stabilization* [2].

In this paper, we investigate another reason why self-stabilizing algorithms may not be more common in practice. We characterize this reason with the term *true distribution* which is borrowed from a seminal paper by Maekawa [11] in the area of solving the distributed mutual exclusion problem. Intuitively, a protocol is truly distributed if no site participating in the protocol bears more responsibility than another one. Usually, true distribution is a very desirable property, since no site may become an availability or performance bottleneck. Many self-stabilizing algorithms are (in this sense) truly distributed. However, truly distributed algorithms are usually more costly in terms of execution time or space than centralized or hierarchical solutions.

Algorithm designers often argue against sacrificing true distribution because lack thereof makes algorithms less generally applicable and differing assumptions must be made about individual network nodes, e.g., about their robustness. In reality, however, the processing elements on which network protocols run are far from homogeneous. The routers within the backbone of the Internet are highly specialized machines with mirrored disks, uninterruptible power supplies,

* Work was supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Emmy Noether programme.

skilled maintenance personnel, and no application load whereas, e.g., standard university workstations are often low-cost PCs with aged hardware and overfull, unbackuped disks. Even the hardware components which are directly concerned with handling network traffic (like switches and bridges) exist in many different forms. Moreover, real network structures are usually hierarchical (hosts attached to subnetworks which are attached to backbone networks) and not a "general graph" (as is assumed in many distributed algorithms). This fact can be exploited by real network protocols, e.g., by using hierarchies in DNS, NTP or employing hierarchical routing. However, true distribution is a conservative guideline for algorithm design: truly distributed algorithms will also run in heterogeneous environments. But truly distributed algorithms cannot exploit the characteristics of these environments to be more efficient.

What is needed are more flexible algorithms, i.e., algorithms which can be adapted or adapt themselves to different network settings. As a first step towards such algorithms, methods can help in which truly distributed algorithms can be adapted to heterogeneous network settings using additional structures. In this paper, we describe such a method for self-stabilizing algorithms in which true distribution can be sacrificed in favor of increased efficiency. Based on experiences in other areas [8, 13], we employ the concept of "adding logical structure" to the system to yield self-stabilizing algorithms with superior time efficiency and fault-containment properties. In particular, we apply this principle to the problem of self-stabilizing spanning tree construction.

Our approach can be briefly characterized as decomposing the entire network into a hierarchy of subnetworks and running individual instances of self-stabilizing spanning-tree algorithms within these subnetworks. The results of these algorithms are then combined in a wrapper algorithm which uses information about the hierarchy to yield a global self-stabilizing spanning tree algorithm with (in favorable settings) a logarithmic stabilization time complexity. We give formal conditions under which the composition is correct. We argue that our design principle makes self-stabilizing algorithms more practical by relating our findings to "real" Internet protocols.

We present the case study of self-stabilizing spanning-tree construction in Sect. 3 and discuss the generality and practical applicability of the approach in Sect. 4. However, in the following section we first provide a small toolbox of different self-stabilizing spanning-tree algorithms with which the general scheme of Sect. 3 may be instantiated.

2 A Toolbox of Self-Stabilizing Spanning Tree Algorithms

In this section we give examples of spanning-tree algorithms for different network settings. For references to more algorithms see the book by Dolev [5].

2.1 System Assumptions

The system is usually modeled as a graph of processing elements (processors, processes, nodes), where the edges between these elements model unidirectional

or bidirectional communication links. In this paper, we denote by n the number of nodes in the system and by N an upper bound on n. Communication is usually restricted to the neighbors of a particular node. We denote by δ the diameter of the network (i.e., the length of the longest unique path between two nodes) and by Δ an upper bound on δ. A network is static if the communication topology remains fixed. It is dynamic if links and network nodes can go down and recover later. In the context of dynamic systems, self-stabilization refers to the time after the "final" link or node failure. The term "final failure" is typical for the literature on self-stabilization: Because stabilization is only guaranteed *eventually*, the assumption that faults eventually stop to occur is an approximation of the fact that there are no faults in the system "long enough" for the system to stabilize. It is assumed that the topology remains connected, i.e., there exists a path between any two network nodes even if a certain number of nodes and links may crash.

Algorithms are modeled as state machines performing a sequence of steps. A step consists of reading input and the local state, then performing a state transition and writing output. Communication can be performed by exchanging messages over the communication channels. But the more common model for communication is that of shared memory or shared registers [5]. It assumes that two neighboring nodes have access to a common data structure, variable or register which can store a certain amount of information. These variables can be distinguished between input and output variables (depending on which process can modify them). When executing a step, a process may read all its input variables, perform a state transition and write all its output variables in a single atomic operation. This is called *composite atomicity* [7]. A weaker notion of a step (called *read/write atomicity* [7]) also exists where a process can only either read or write its communication variables in one atomic step. A related characteristic of a system model is its execution semantics. In the literature on self-stabilization this is encapsulated within the notion of a *scheduler* (or *daemon*) [4]. Under a central daemon, at most one processing element is allowed to take a step at the same time.

The individual processes can be *anonymous*, meaning that they are indistinguishable and all run the same algorithm. Often, anonymous networks are called *uniform* networks [7]. A network is *semi-uniform* if there is one process (the root) which executes a different algorithm [7]. While there is no way to distinguish nodes, in uniform or semi-uniform algorithms nodes usually have a means of distinguishing their neighbors by ordering the incoming communication links. In the most general case it is assumed that processes have globally unique identifiers.

Two kinds of spanning trees may be distinguished: *breadth-first search* (BFS) trees result from a breadth-first traversal of the underlying network topology [10]. Similarly, *depth-first search* (DFS) trees are obtained from a depth-first traversal.

2.2 Lower Bounds

The usual time-complexity measure for self-stabilizing algorithms is that of *rounds* [9]. In synchronous models algorithms execute in rounds, i.e., processors execute steps at the same time and at a constant rate. Rounds can be defined in asynchronous models too, where the first round ends in a computation when every processor has executed at least one step. In general, the i-th round ends, when every processor has executed at least i steps so communication between any two processors in a particular system takes at least $\Omega(d)$ rounds. This is because it normally takes at least one round to propagate information between two adjacent processors. For the case of self-stabilizing spanning-tree construction and under certain assumptions, an arbitrary initial state may make it necessary to propagate information through the entire network. Therefore, a general lower bound of $\Omega(d)$ rounds can be assumed for self-stabilizing spanning-tree algorithms.

2.3 Two Basic Algorithms

The algorithm by Dolev, Israeli and Moran. One of the first papers to appear was by Dolev, Israeli and Moran [6, 7] in 1990. It contains a self-stabilizing BFS spanning-tree construction algorithm for semi-uniform systems with a central daemon under read/write atomicity. In the algorithm, every node maintains two variables: (1) a pointer to one if its incoming edges (this information is kept in a bit associated with each communication register), and (2) an integer measuring the distance in hops to the root of the tree. The distinguished node in the network acts as the root. The algorithm works as follows: The network nodes periodically exchange their distance value with each other. After reading the distance values of all neighbors, a network node chooses the neighbor with minimum distance *dist* as its new parent. It then writes its own distance into its output registers, which is $dist + 1$. The distinguished root node does not read the distance values of its neighbors and simply always sends a value of 0.

The algorithm stabilizes starting from the distinguished root node. After sufficient activations of the root, it has written 0 values into all of its output variables. These values will not change anymore. Note that without a distinguished root process the distance values in all nodes would grow without bound. More specifically, after reading all neighbors values for k times, the distance value of a process is at least $k + 1$. This means, that after the root has written its output registers, the direct neighbors of the root—after inspecting their input variables—will see that the root node has the minimum distance of all other nodes (the other nodes have distance at least 1). Hence, all direct neighbors of the root will select the root as their parent and update their distance correctly to 1. This line of reasoning can be continued incrementally for all other distances from the root. Hence, after $O(\delta)$ update cycles the entire tree will have stabilized.

The algorithm by Afek, Kutten and Yung. In the same year as Dolev, Israeli and Moran [6] published their algorithm, Afek, Kutten and Yung [1] presented

an self-stabilizing algorithm for a slightly different setting. Their algorithm also constructs a BFS spanning-tree in the read/write atomicity model. However, they do not assume a distinguished root process. Instead they assume that all nodes have globally unique identifiers which can be totally ordered. The node with the largest identifier will eventually become the root of the tree.

The idea of the algorithm is as follows: Every node maintains a parent pointer and a distance variable like in the algorithm above, but it also stores the identifier of the root of the tree which it is supposed to be in. Periodically, nodes exchange this information. If a node notices that it has the maximum identifier in its neighborhood, it makes itself the root of its own tree. If it learns that there is a tree with a larger root identifier nearby, it joins this tree by sending a "join request" to the root of that tree and receiving a "grant" back. The subprotocol together with a combination of local consistency checks ensures that cycles and fake root identifiers are eventually detected and removed.

The algorithm stabilizes in $O(n^2)$ asynchronous rounds and needs $O(\log n)$ space per edge to store the process identifier. The authors argue this to be optimal since message communication buffers usually communicate "at least" the identifier.

2.4 Summary

This section has presented two self-stabilizing spanning tree construction algorithms. The one by Afek, Kutten and Yung [1] can be characterized as truly distributed. The semi-uniform algorithm of Dolev, Israeli and Moran [6, 7] takes a first step towards exploiting heterogeneity: it is therefore simpler than the other algorithm but must make additional robustness assumptions. Both algorithms can be used as building blocks for the method described in Sect. 3.

3 Adding Structure for Constructing Efficient Self-Stabilizing Spanning-Tree Algorithms

In this section we show how sacrificing full distribution can help to improve the efficiency of self-stabilizing algorithms. To demonstrate this, we perform a case study of applying the general principle of "introducing structure" to the area of self-stabilizing spanning-tree construction. By doing this, it is possible to transform an arbitrary self-stabilizing spanning-tree algorithm into one with increased efficiency and fault-containment properties.

3.1 System Assumptions and Base Algorithm Interface

We model a distributed system as a connected graph $G = (\Pi, E)$, where $\Pi = \{P_1, \ldots, P_n\}$ is the set of processing elements and $E \subseteq V \times V$ is the set of communication links between the processing elements. The starting point for the transformation is a set of self-stabilizing spanning-tree construction algorithms (e.g., those presented in Sect. 2). The remaining system assumptions (shared

memory/message passing, distinguished identifiers/distinguished root, etc.) are the minimum assumptions necessary in order for the algorithms to work.

Within the transformation, we assume that there are multiple instances of the algorithm running concurrently throughout the network. We denote the set of all instances by $I = \{A, B, C, \ldots\}$. Each instance X of the algorithm consists of a set of local algorithm modules $x, x', \ldots$, one for each processing element which participates in the algorithm instance X. To simplify notation, x_i always denotes the local module of instance X running in P_i (e.g., the module of X running on P_3 is denoted x_3). There is at most one module of X running per processing element. Additionally, every processing element participates in at least one algorithm instance. We capture the distribution of algorithm instances to processing elements using a function $d : I \to 2^{\Pi}$ (where 2^{Π} denotes the powerset of Π, i.e., the set of all subsets of Π). We assume that the set of all algorithm instance names can be totally ordered (e.g., by using an alphabetic ordering).

Every algorithm instance running on a set of processing elements has a set of *possible roots*. For example, semi-uniform algorithms have only one preconfigured root. In other algorithms (like the one by Afek, Kutten and Yung [1]), the root will be the processing element with maximal identifier. In the case of process crashes, the processing element with the second highest identifier will eventually become root. Given the assumption that at most 1 process can fail by crashing within the protocol instance, it makes sense to select at least two processing elements, namely the ones with the highest identifiers, as possible roots for that algorithm instance. We capture the assignment of algorithm instances to possible roots using a function $r : I \to 2^{\Pi}$.

We assume that the individual instances of the algorithm do not interfere with each other during execution, i.e., each instance is able to execute as if it were the only algorithm running in the network. We assume that the communication topology needed by an algorithm instance respects the given communication topology, i.e., the communication graph of the algorithm instance is a subgraph of G.

We are not concerned how individual algorithm instances are created, we merely assume that they are up and running and that every processing element P_i has a means to access the interface of each algorithm module that is running on P_i. For example, bootstrapping of the algorithm can be performed off-line or on-line by an administrator who configures and starts the local algorithm modules as individual processes on the processing elements. Overall, we capture the deployment of algorithm instances in the definition of a *system*.

Definition 1 (System). *A* system $S = (\Pi, I, d, r)$ *consists of a set of processing elements* $\Pi = \{P_1, \ldots, P_n\}$*, a set of algorithm instances* I*, a function* $d : I \to 2^{\Pi}$*, and a function* $r : I \to 2^{\Pi}$*, such the following holds:*

1. *Every processing element participates in at least one algorithm instance, formally:*

$$\forall P \in \Pi : \exists X \in I : P \in d(X)$$

2. *A possible root P_i of an algorithm instance X always runs a local module x_i, formally:*

$$\forall X \in I : r(X) \subseteq d(X)$$

The basic interface of each module x_i of algorithm instance X consists of two methods:

- **boolean** *is_root*()
  This method returns `true` if and only if the processing element on which x_i runs is considered to be the root of the spanning tree which is maintained by algorithm instance X.
- **Neighbors** *parent*()
  If the processing element P on which x_i runs is not the root of the spanning tree which is maintained by algorithm instance X, then this method will return the identifier of the processing element P' which is considered to be the parent in that spanning tree. The identifier P' is guaranteed to be the identifier of a neighbor of P.

The algorithm interface is general enough to encapsulate any of the spanning-tree construction algorithms presented in Sect. 2.

As an example, consider the two system setups in Fig. 1. There are three instances A, B and C of the algorithm running on a network of five processing elements $P_1, \ldots, P_5$. For example, on the left side of Fig. 1 algorithm instance C runs on two processing elements P_3 and P_5 and consequently has two local modules called c_3 and c_5. Note that algorithm instances may overlap and that every processing element is covered by at least one algorithm instance.

An application running on some processing element P_i may access the interface of any algorithm module running on P_i. For example, an application running on P_3 may access b_3 and c_3.

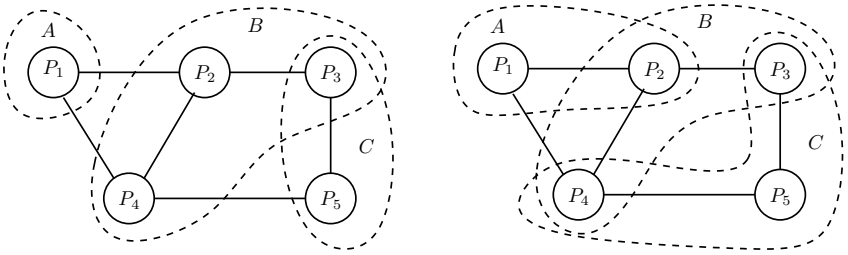

Fig. 1. Two simple system setups with three algorithm instances A, B, and C on a network of five processing elements $P_1, \ldots, P_5$.

3.2 Introducing Structure

We now define an additional "overlay" structure $H = (I, \leftarrow)$ which is a rooted tree over I. The $\leftarrow$ relation is the "parent" relation over I, i.e., a child node al-

ways points to the parent node in $\leftarrow$. Additionally, H must satisfy two additional properties, *connectedness* and *consistency*. We first define connectedness.

Definition 2 (Connectedness). *A* structure $H = (I, \leftarrow)$ is connected with respect to some system $S = (\Pi, I, d, r)$ *if and only if the following holds: If* $X \leftarrow Y$ *in* H *then* $d(X)$ *and* $d(Y)$ *share a processing element in* S*, formally:*

$$\forall X, Y \in \Pi : X \leftarrow Y \Rightarrow d(X) \cap d(Y) \neq \emptyset$$

Basically, connectedness means that any pair of algorithm instances that are connected in H have a processing element in common. Sometimes it is impossible to find a connected structure for a system. This is the case for the system on the left side of Fig. 1 because A does not overlap with any other algorithm instance.

Fig. 2 gives examples and counterexamples of connected structures: (a) and (b) are connected structures with respect to the system on the left side of Fig. 2. The structure (c) however is not connected since $A \leftarrow C$ holds but A and C do not share a common processing element.

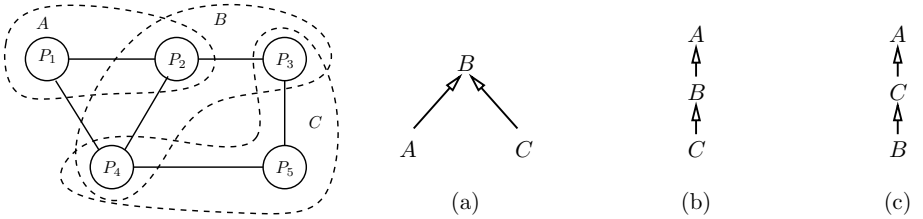

Fig. 2. Examples and counterexamples of connected structures for the example setting on the left: examples (a) and (b) satisfy the definition, (c) not.

Definition 3 (Consistency). *A* structure $H = (I, \leftarrow)$ is consistent with respect to a system $S = (\Pi, I, d, r)$ *if and only if the following holds: If algorithm instance* X *is a parent of algorithm instance* Y *in* H*, then every possible root of* Y *must run a local algorithm module of* X*, formally:*

$$\forall X, Y \in I : X \leftarrow Y \Rightarrow r(Y) \subseteq d(X)$$

Intuitively, consistency means that it is possible to "go up" in the structure H at any root of a tree established in some algorithm instance. As an example, consider the left side of Fig. 2 and structure (a) and assume that the possible root of B is P_2 (i.e., $r(B) = \{P_2\}$). This setting is consistent, since $A \leftarrow B$ holds and every possible root of B is a node running a local module of instance A. On the other hand, if $r(B) = \{P_3\}$, then the structure would be inconsistent for the given system, since P_3 is not participating in algorithm instance A.

We assume that every processing element has means to access the structure H. In particular, a processing element can determine whether some algorithm instance is the root of H.

3.3 Deriving a Global Spanning Tree

Given the set of algorithm instances and the additional structure H, we now want to derive a self-stabilizing algorithm that constructs a spanning tree over the entire set of processing elements. Basically, we want to implement the spanning-tree interface given in Sect. 3.1 (i.e., the *is_root*() and the *parent*() methods).

First consider implementing the *is_root*() operation. Basically, the root of the entire spanning tree is the root of the top-level algorithm instance. Fig. 3 depicts the code of an implementation on a particular processing element P_i.

```
boolean is_root()
begin
  X := ⟨root element of H⟩
  if P_i ∈ d(X) then
    return x_i.is_root()
  else
    return false
  end
end
```

Fig. 3. Implementing *is_root*() on processing element P_i.

Now, consider implementing the *parent*() operation. The idea of the implementation is to take the "smallest" level algorithm instance (with respect to H) and return the parent pointer of that instance. Note that "smallest" is not always well-defined since $\leftarrow$ is a partial order. In case there are two incomparable algorithm instances running on the same processing element, we use the assumed alphabetical order between instance names as a tie breaker. Along this line of reasoning, we now define a total order on the set of algorithm instances.

Definition 4. *Given a set I of algorithm instances and a structure $H = (I, \leftarrow)$. Let $\preceq$ denote the reflexive, transitive closure of $\leftarrow$. Define the* relation $\leq$ over $I \times I$ *as follows: $X \leq Y$ holds iff one of the following two cases is true:*

1. *either $X \preceq Y$,*
2. *or if $\neg X \preceq Y$ and $\neg Y \preceq X$, then X is "alphabetically smaller" than Y.*

Using $\leq$, we can implement *parent*() as shown in Fig. 4. We note that it is also possible to implement *parent*() by choosing the largest algorithm instance instead of the smallest (with respect to $\leq$) without compromising the correctness of our approach. If the largest instance is preferred, then the parent pointers of lower level algorithm instances dominate those of higher level instances and lower level structures are maintained in the combined tree rather than higher level structures. This might be preferable in some situations because it preserves a form of locality. If higher level pointers are preferred, then the three will generally be "less tall", i.e., the average distance to the root will be shorter.

Neighbors *parent*()
begin
  $X :=$ ⟨smallest algorithm instance with respect to $\leq$ running on P_i⟩
  return $x_i.parent()$
end

Fig. 4. Implementing *parent*() on processing element P_i.

3.4 Examples

Fig. 5 shows an example system with a connected and consistent structure. The possible roots of the algorithm instances are also given in the figure. The overall spanning tree which is eventually constructed and emulated by our algorithm results from "overlaying" the individual spanning trees constructed within the algorithm instances.

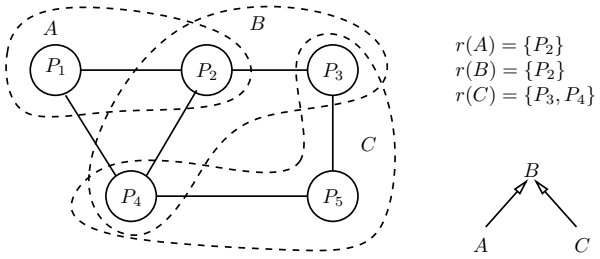

Fig. 5. Example system with a connected and consistent structure together with the possible roots of the algorithm instances.

For example, algorithm instances A and B could be running with any (semi-uniform) self-stabilizing spanning tree algorithm which has P_2 pre-configured as the root. Instance C could be running an algorithm whose root eventually is the processing element with highest identifier. In this case, P_3 may have the highest identifier and P_4 the second highest. This makes sense if one of the two processes may crash. If P_3 crashes, the spanning tree constructed by instance C will stabilize to P_4 as root. Without a crash, the root will be P_3. The trees computed by the three algorithm instances are shown on the left side of Fig. 6. The "overlay" spanning tree computed by our combined algorithm is depicted on the right side of Fig. 6. Note that at processing element P_4 there are two parent pointers available (one in algorithm instance B and one in C). In such cases, the pointer of the "smaller" instance with respect to $\leq$ is chosen (i.e., B).

Fig. 7 (left) shows another example with nine processing elements and four algorithm instances with their local spanning trees. Instance B forms the "backbone" of the network and is the top level instance. The resulting global spanning tree is depicted on the right side of Fig. 7. Note that the parent pointer of P_7

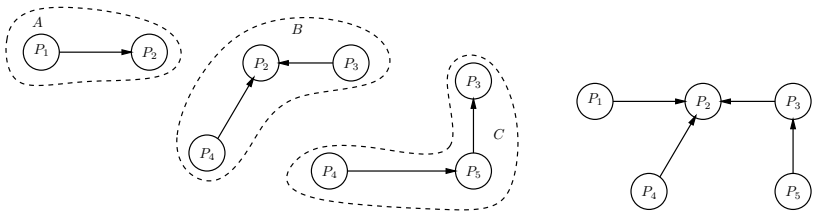

Fig. 6. Global spanning tree resulting from the example in Fig. 5: partial trees (left) and global tree (right).

in the global tree is the parent pointer from instance B and hence points to P_8 (and not the pointer of instance C because it ranks less according to $\leq$).

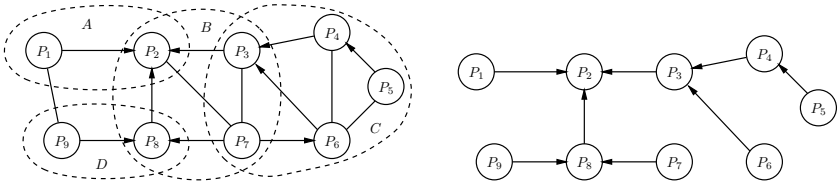

Fig. 7. Example with a "backbone" algorithm instance B and three "subnetwork" instances A, C, D (left) and resulting tree (right).

3.5 Correctness of the Transformed Algorithm

We now argue that the implementation of the spanning-tree operations given above in fact results in a self-stabilizing spanning-tree algorithm for the entire network provided the underlying algorithm instances are self-stabilizing and the structure is connected and consistent.

Assuming the underlying spanning-tree algorithm instances have all stabilized, we now give a mathematical definition of the ordering which is emulated by the implementation sketched above and show that it is in fact a rooted spanning tree.

Lemma 1. *Given a system $S = (\Pi, I, d, r)$ and a connected and consistent structure $H = (I, \leftarrow)$. If all algorithm instances $X \in I$ have stabilized to rooted spanning trees $T_X = (d(X), \overset{X}{\leftarrow})$, then the implementation given in Sect. 3.3 emulates the following rooted global spanning tree $T = (\Pi, \leftarrow)$:*

$$P_i \leftarrow P_j \Leftrightarrow \exists X \in I : \left[P_i, P_j \in d(X) \wedge P_i \overset{X}{\leftarrow} P_j \wedge [\forall Y \in I : X \leq Y]\right]$$

Proof. Basically, H determines the basic skeleton of T. The individual spanning trees of the algorithm instances yield the substructures of T. The property of

consistency guarantees that the T is in fact a tree. The fact, that S is a system implies that every processing element is part of some instance in H. This together with the property of connectedness implies that every node is part of T.

It remains to be shown that the overall algorithm stabilizes, given that the underlying algorithm instances stabilize.

Lemma 2. *Given a system $S = (\Pi, I, d, r)$. Starting from an arbitrary initial state, if all algorithm instances $X \in I$ stabilize to a rooted spanning tree, then the overall algorithm will eventually stabilize to a rooted spanning tree.*

Proof. Since all algorithm instances do not interfere with each other, each algorithm instance will stabilize independently. Take the time t as the maximum stabilization time of any $X \in I$. After t, the overall algorithm will return a rooted spanning tree according to Lemma 1.

Theorem 1. *Given a system $S = (\Pi, I, d, r)$ and a connected and consistent structure $H = (I, \leftarrow)$. If all algorithm instances $X \in I$ run self-stabilizing spanning tree algorithms, then the implementation given in Sect. 3.3 emulates a global self-stabilizing spanning tree.*

3.6 Analysis

The composed algorithm has two main advantages. The first advantage is improved stabilization time. Consider the case where all algorithm instances employ a time-optimal self-stabilizing spanning tree algorithm (such as the one by Dolev, Israeli and Moran [7]). This algorithm stabilizes in $O(\delta)$ rounds where δ is the diameter of the network on which the algorithm runs. The point to note is that all algorithm instances run in parallel and that the diameters of the subnetworks in which the algorithm instances run can be considerably smaller than the overall diameter δ. Let $\delta_1, \ldots, \delta_k$ denote the diameters of the algorithm instances. Then the proof of Lemma 2 shows that the stabilization time is $O(\max_{1 \leq i \leq k} \delta_i)$. Note that this improvement is a result from the unique way in which self-stabilizing algorithms may be composed in parallel.

Consider a symmetric hierarchical decomposition of the network into subnetworks of equal size and a three level hierarchy as depicted in Fig. 8. The number of nodes in the network is $3^3 = 27$ and the network diameter is $\delta = 5$. But the overall algorithm will stabilize in steps proportional to the diameter of the largest instance network, which is 1. In general, an optimal decomposition into algorithm instances using such a hierarchic decomposition can improve the stabilization time in logarithmic scale. Decomposing n nodes into k levels yields a stabilization time in the order of $\log_k n$. Of course, this is not always feasible because it depends on the network topology, but at least sub-optimal results are achievable in practice since real networks like the Internet are often hierarchic and the diameter of the largest subnetwork is much smaller than the total diameter.

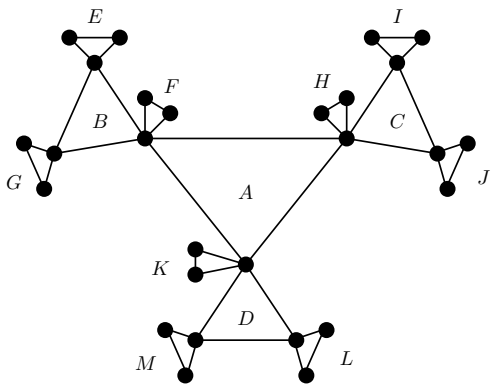

Fig. 8. Hierarchical decomposition of 27 nodes in a three layer hierarchy. The top layer consists of algorithm instance A, the middle layer of B, C and D, and the bottom layer of E, F, G (below B), H, I, J (below C) and K, L, M (below D).

The example shows that there are two extreme cases of the structure $H = (I, \leftarrow)$. The one case arises when I consists of just a single algorithm instance. In this case, adding structure does not have any benefit at all. The other case arises when I contains exactly one algorithm instance for every pair of neighboring nodes in H. In this case, the spanning tree is directly determined by H.

The second advantage of the composed algorithm is fault-containment. By adding the fixed structure, faults can only lead to perturbations within the algorithm instances in which they happen. This is in contrast to the case where a standard spanning-tree algorithm runs in the entire network: a single fault (e.g., of the root) can lead to a global reconfiguration. However, the level of fault-containment again depends on the distribution of algorithm instances to processing elements: if one processing element participates in all algorithm instances then a failure of this node may also cause global disruption.

4 Discussion

Many protocols running in the Internet like DNS or NTP sacrifice true distribution to gain efficiency, but still work rather robustly in practice. Their robustness stems from their ability to adapt their internal structures to the hierarchical and heterogeneous structure of the Internet. One particularly instructive example is the area of routing protocols.

When the Internet was a small network, routers used a truly distributed protocol to update their routing tables. Every router periodically broadcasted information to *all* other routers in the system and incorporated the received data into its own table (this was called *link-state routing* [12]). When the Internet evolved as an increasing collection of independent subnetworks (called *autonomous systems*) using a set of routers connected by high bandwidth communication links (the *backbone*), link-state routing was considered to be too

inefficient for the entire network. The hierarchical nature of the Internet is exploited through modern routing protocols: within autonomous systems modern link-state routing strategies like OSPF [12] are applied (interior gateway routing). These are different from those running on the backbone (exterior gateway routing, e.g., BGP [14]).

Today, the standard approach to realize local area subnetworks, is to use switched Ethernets. In this case, it is also possible to increase the robustness by allowing for path redundancy between switches. In general, loops in the network topology of an Ethernet may cause data duplication and other forms of confusion. However, modern switches incorporate an adaptive spanning tree protocol to establish a unique path between all switches [3]. If a network segment becomes unreachable or network parameters are changed, the protocol automatically reconfigures the spanning-tree topology by activating a standby path.

The spanning-tree protocol employed in switches has striking similarities to some of the protocols described in Sect. 2. Initially, switches believe they are the root of the spanning tree but do not forward any frames. Governed by a timer, they regularly exchange status information. These messages contain (1) the identifier of the transmitting switch (usually a MAC address), (2) the identifier of the switch which is believed to be the root of the tree, and (3) the "cost" of the path towards the root. Using this information, a switch chooses the "shortest" path towards the root. If there are multiple possible roots, it selects the root with the smallest identifier (lowest MAC address). Links which are not included in the spanning tree are placed in blocking mode. Blocking links do not forward data frames but still transport status information.

The presentation above shows that the Internet maintains a set of protocols in a hierarchy to increase efficiency. Routing protocols contain spanning-tree construction, and so the hierarchical deployment is very similar to the approach sketched in Sect. 3. Algorithm instances can be replaced by self-stabilizing versions, so—apart from increasing efficiency—the hierarchical approach makes a transparent migration towards a wider deployment of self-stabilizing algorithms feasible.

5 Conclusions

We have presented a method to construct self-stabilizing algorithms with superior stabilization time and fault-containment properties from given solutions. The price we pay is sacrificing true distribution. We have argued that this is not a high price to pay in practice since real networks are not homogeneous. The only problem is to find flexible methods to map logical algorithm structures to physical network structures. We claim that our approach is one solution to this problem and hence may help to make self-stabilizing algorithms more practical.

The approach is general because it is applicable in almost the same way to many other important problems including the problem of self-stabilizing leader election and mutual exclusion. However, determining how to map the logical

structures of these algorithms to existing network structures is a non-trivial design task and needs to be investigated further.

References

1. Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In J. van Leeuwen and N. Santoro, editors, *Distributed Algorithms, 4th International Workshop*, volume 486 of *LNCS*, pages 15–28, Bari, Italy, 24–26 Sept. 1990. Springer, 1991.
2. J. E. Burns, M. G. Gouda, and R. E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7:35–42, 1993.
3. Cisco Systems Inc. Using VlanDirector system documentation. Internet: `http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/sw_ntman/cwsimain/cwsi2/cwsiug2/vlan2/index.htm`, 1998.
4. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Comm. of the ACM*, 17(11):643–644, 1974.
5. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
6. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In C. Dwork, editor, *Proceedings of the 9th Annual ACM Symp. on Principles of Distribted Computing*, pages 103–118, Québec City, Québec, Canada, Aug. 1990. ACM Press.
7. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
8. L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In B. Magnusson, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309–333, Malaga, Spain, June 2002. Springer.
9. C. Génolini and S. Tixeuil. A lower bound on dynamic k-stabilization in asynchronous systems. In *Proc. 21st Symposium on Reliable Distributed Systems (SRDS 2002), IEEE Computer Society Press*, pages 211–221, 2002.
10. D. E. Knuth. *The Art of Computer Programming*, volume III (Sorting and Searching). Addison-Wesley, Reading, MA, second edition, 1997.
11. M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.
12. J. Moy. OSPF version 2. Internet: RFC 1583, Mar. 1994.
13. H. Pagnia and O. Theel. Sacrificing true distribution for gaining access efficiency of replicated objects. In *Proc. 31st IEEE Hawaii Intl. Conference on System Sciences (HICSS-31)*, Big Island, HI, USA, 1998.
14. Y. Rekhter. A border gateway protocol 4 (BGP-4). Internet: RFC 1771, Mar. 1995.