

## Self-Stabilizing Distributed Systems & Sensor Networks

Z. Shi and Pradip K Srimani

June 7, 2004



# Contents

- 1 Self-Stabilizing Distributed Systems & Sensor Networks 1**
- 1.1 Distributed Systems . . . . . 1
- 1.2 Fault Tolerance in Mobile Distributed Systems & Self-Stabilization . . . . . 3
  - 1.2.1 Requirements of Self-Stabilization . . . . . 4
  - 1.2.2 Implementation Issues . . . . . 5
  - 1.2.3 Classifications of Self-Stabilizing Systems . . . . . 6
- 1.3 Sensor Network . . . . . 8
  - 1.3.1 Unbounded Asynchronous Unison [Her03b] . . . . . 8
  - 1.3.2 Bounded Unison . . . . . 9
- 1.4 Maximal Independent Set . . . . . 11
- 1.5 Minimal Domination . . . . . 12
- 1.6 Neighborhood Identification . . . . . 14
- 1.7 Neighborhood Unique Naming . . . . . 14
- 1.8 Acknowledgement . . . . . 15



# Chapter 1

## Self-Stabilizing Distributed Systems & Sensor Networks

The purpose of this chapter is to provide a brief description of the basic models of self-stabilization that have been used in designing fault tolerant distributed algorithms; and then, to detailed description of models for sensor networks [Her03b], P2P networks [DGRS03] and cooperative mobile agents [Gho01] and (3) to critically evaluate how efficiently the standard models of self-stabilization can be used to design fault-tolerant protocols for mobile environments [Her03b, DGRS03, Gho01] in presence of mobile clients (along with their constraints like low battery power, unreliable communication medium, frequent disconnection and reconnection characteristics. We show how the standard models of self-stabilization can be modified to accommodate the requirements of a sensor network and provide a couple of example protocols. We also explore designing protocols for some simple global primitives using the modified models [GHJS03].

### 1.1 Distributed Systems

A distributed system essentially consists of a number of *autonomous* computers (with their own hardware and software) that are connected to each other to form a communication network. Even

## 2 CHAPTER 1. SELF-STABILIZING DISTRIBUTED SYSTEMS & SENSOR NETWORKS

though it may be connected to other computers, each computer can run its applications independently of the others, while the networking software (and hardware) provides the functionalities as database systems, and shared storage (data and code). Typically, there is no shared memory and communication between two connected nodes (sites) is implemented through explicit message passing. An important characteristic of the system is that it presents to the user, at any of the participating sites, a view of a *single*, highly reliable program, thus making the decentralized nature of the hardware and the software transparent to the user. The challenge of developing software, hardware and network interfaces becomes more formidable because of the need to dynamically respond to failures and subsequent recoveries.

In the last decade we have seen excellent growth in the areas of client server and networked personal computing systems. In the next phase of distributed computing, we will probably see a growth in the area of mobile computing systems where a very close coupling (coordination) is necessary between a large number of applications running on different platforms across the globe, where the computing nodes are no longer stationary in space and they frequently enter and leave the system. In addition to mobile computing systems in general, large scale sensor networks, cooperative peer-to-peer (P2P) networks, and cooperating mobile agents are all example settings where distributed computing is indispensable. Cellular networks and sensor networks consist of a collection of a large number of identical relatively low-powered nodes, with limited communication and computing capabilities. An important characteristic is that the nodes are distributed at random, or even with some a-priori planning, but at any time some of them may not be available, and there is only one channel for message transmission between nodes and base stations. A peer-to-peer network is a dynamic, and scalable, set of nodes (peers), where the objective is to distribute the cost of storing and sharing large amounts of data. “Each peer can join or leave the system at any moment, and can communicate with any other peer under the only hypothesis that the two peers are aware of each other. The main characteristics of peer-to-peer systems are the ability to pool together and harness a large amount of resources, self-organization, load-balancing, adaptation and fault-tolerance” [GRS03].

The communication network architecture is a key component in a distributed system and its characteristics are fundamental to the performance of the applications running on that system. In the past, the relative performance of the communication system has lagged behind that of the

## 1.2. FAULT TOLERANCE IN MOBILE DISTRIBUTED SYSTEMS & SELF-STABILIZATION<sup>3</sup>

computing engine, but the advent of fiber optics and gigabit networking have started to close the gap. The class of applications that can be cost effectively implemented on distributed systems has broadened, and the intra-nets are now more popular than ever before.

### 1.2 Fault Tolerance in Mobile Distributed Systems & Self-Stabilization

Two of the most desirable attributes of the modern mobile distributed systems are fault tolerance and scalability. We need systems designed so that they can recover from transient faults spontaneously and reconfigure (scale) themselves without any need for external intervention.

A common approach to designing fault tolerant systems is to mask the effects of the fault. However, fault masking is not free; it requires additional hardware or software and it considerably increases the cost of the system. This additional cost may not be an economic option, especially when most faults are transient in nature, and a temporary unavailability of a system service is acceptable. *Self-stabilization* is a relatively new way of looking at system fault tolerance, which provides a “built-in-safeguard” against “transient failures” that might corrupt the data in a distributed system.

The objective of self-stabilization is (as opposed to masking faults) to recover from faults in a reasonable amount of time and without intervention by any external agent. Self-stabilization is based on two basic ideas: first, the code executed by a node is re-entrant, incorruptible (as if written in a fault resilient memory) and transient faults affect only data locations; second, a fault free-system behavior is usually checked by evaluating some predicate of the system state variables. The checking oracle may be complex but is not part of the self-stabilizing algorithm design. Every node has a set of local variables, whose contents specify the local state of the node. The state of the entire system, called the *global state*, is the union of the local states of all the nodes in the system. Each node is allowed to have only a partial view of the global state, and this depends on the connectivity of the system and the propagation delay of different messages. The objective in a distributed system is to arrive at a desirable global final state called a *legitimate state*.

### 1.2.1 Requirements of Self-Stabilization

There exist several models for self-stabilization; we present here only the basic common concepts of these models. The *state* of a node is specified by its local variables. The system state is a vector of all local states of the participating nodes. We use  $\mathcal{T}$  to denote the set of all possible system states. A system state is either *legitimate* or *illegitimate*. The precise specification of a legitimate state depends on the algorithm, but as a general rule, when the system is in a legitimate it has the property required by that application. To allow system recovery after transient faults, each node executes repeatedly a piece of code. This code consists of a set of rules:

```

begin
    rule
    :
    rule
end

```

Each rule has the form:

```
(label) [guard]: <program>;
```

A guard is a boolean expression of the variables that the processor can read: its own variables and the variables of its neighbors. The program part of a rule is the description of the algorithm used to compute the new values for local variables. If the guard of a rule is true, that rule is called *enabled*. When at least one rule is enabled the node is *privileged*. An *execution* of a enabled rule is the determination of the new node state value using the algorithm described by the program part of the rule. A *move* of a node is the execution of a nondeterministically chosen enabled rule.

In other words, there is a relation  $\mathcal{R} \subset \mathcal{T} \times \mathcal{T}$  such that if  $(s_i, s_f) \in \mathcal{R}$ , then (i) the states  $s_i, s_f$  differ by a single node  $x$  value, and (ii) if the system is in state  $s_i$  there is a enabled rule of node  $x$  such that after execution of the corresponding code, the system is in state  $s_f$ . A system evolution  $\mathcal{E} = (s_i)_{i \in I}$ , is a finite or infinite sequence of moves such that (i) if  $(s_i, s_{i+1})$  is a consecutive pair of states in  $\mathcal{E}$  then  $(s_i, s_{i+1}) \in \mathcal{R}$ , and (ii) if the system evolution has a finite number of states,  $s_f$  being the last one, then there is no state  $s \in \mathcal{T}$  such that  $(s_f, s) \in \mathcal{R}$ .

To prove the correctness of a self-stabilizing algorithm, the conditions of closure and the convergence must be shown. The closure property means that when the system is in a legitimate state

## 1.2. FAULT TOLERANCE IN MOBILE DISTRIBUTED SYSTEMS & SELF-STABILIZATION<sup>5</sup>

the next state is always a legitimate state. The convergence property means that for any state and for any sequence of possible moves, after a finite number of moves the system is in a legitimate state. As Gouda observes in his paper [Gou95], self-stabilization can be in principle defined by a set,  $\mathcal{T}$ , a relation  $\mathcal{R} \subset \mathcal{T} \times \mathcal{T}$  and a specification of the legitimate state set  $\mathcal{L}$ . Different classes of closure and convergence can be defined and general methods of proving the self-stabilization can be sketched.

One useful and elegant strategy to prove the correctness of self stabilizing algorithms is to use bounded monotonically decreasing functions defined on global system states [Kes88]; some existing self-stabilizing algorithms are proved to be correct by defining a bounded function that is shown to decrease monotonically at every step [Hua93]. Many self-stabilizing algorithms do not use this bounded function method since it is usually very very difficult to design such a function. In stead, they develop a different proof technique using induction on the number of nodes in the graph.

### 1.2.2 Implementation Issues

The stabilizing algorithms achieve fault tolerance in a manner that is radically different from traditional fault tolerance in distributed systems. The paradigm allows us to abandon failure models and a bound on the number of failures. The theory is elegant, but how practical is the concept for implementation with present day technology? Here are some issues:

- The concept of the global state of the system requires a *common* time for all nodes. The physical time may be used as a common time but it may not be explicitly used by the component processes (drifts in local clocks, relativity etc.) The partial order relation (among events) generated by message exchange can not be uniquely extended to a total order relation. If there is no global clock, global states can not be defined and legitimate states must defined locally, i.e. based on the local state of a node and the states of its neighboring nodes and on the partial order relation associated to sending and receiving messages. This greatly complicates the correctness proof of any stabilizing algorithm.
- A *move* is a complex operation; the state of the neighbors must be read, the guards must be

## 6 CHAPTER 1. SELF-STABILIZING DISTRIBUTED SYSTEMS & SENSOR NETWORKS

evaluated and the associated code segment must be executed. Some models assume that a new move may not start until the previous move is completed, i.e. the moves are atomic. In a real distributed system the reading of a neighbor's state can be implemented in two ways: one option is to request every node to send its state to his neighbors periodically or whenever it changes its state. Each node caches the state received from its neighbors and moves according to the state cached in its memory. The other option is to use a query message: when a node needs to read its neighbor's state, it sends a query message and waits for reply. In both cases, when a node moves, it uses the cached states of the neighbors instead the real states of its neighbors. Since the states of the neighbors may have already been changed, the moves are not really atomic.

- To enforce the atomicity and the serializability of the moves Dijkstra, [Dij74], has introduced the concept of central daemon. When multiple nodes are privileged, the central daemon arbitrarily selects one node to be active next. The concept of a central daemon is very much against the concept of a distributed system in that it serializes the moves and does not allow concurrent node executions. Proving correctness is easier for a serial execution, but there are many parallel executions that might be "equivalent" with a serial execution [BHG87]. These executions should be allowed by an efficient move scheduler.

Different models of self-stabilization offer different prospects of cost effective implementation of the concept and it is not clear at this point which would win.

### 1.2.3 Classifications of Self-Stabilizing Systems

A *model* can be viewed as an *interface definition* and a set of assumptions (the algorithm designer can make) that define the behavior of the system. Unfortunately, there is no unifying model for the distributed system concept [ST96]. If some specific features of a system are not introduced in the model, the algorithms may be less efficient than they could possibly be; on the other hand, those features may not general enough to be present in all systems.

Self-stabilizing algorithms have been designed for two interprocess communications paradigms: shared memory and message passing. Since we are interested in self-stabilizing algorithms for dis-

## 1.2. FAULT TOLERANCE IN MOBILE DISTRIBUTED SYSTEMS & SELF-STABILIZATION

tributed systems, we assume the message passing paradigm. It should be observed that a lower level protocol that sends and receives messages may transform a message passing model into a shared memory one.

### **Anonymous vs Id-based Networks**

The concept of *node identity* is important in designing distributed algorithms. If each node has a unique hardwired *id*, the network is *id*-based; otherwise the network is *anonymous*. The anonymous network is a weaker model than an *id*-based network. For some problems there are no deterministic algorithms in anonymous networks [Ang80]. The impossibility stems from the lack of deterministic symmetry breaking mechanisms without unique ids. In general, it is far more difficult to design algorithms for anonymous networks than for *id*-based networks. The *id*-based network is a more realistic model to design self-stabilizing algorithms, but a database of the used *id*'s is necessary to be maintained by a central authority; the addition of a new node requires a database search and the assignment of a new distinct *id*. This concept has been used in practice for a while (Ethernet addresses and IP addresses) and it has proved to be convenient. In principle, each node in an *id*-based network may have a global information of the topology and the state of all other nodes. Hence a local algorithm may be used to solve the problem. This scheme may be unacceptable since the information needed to update each node state is large and takes a considerable bandwidth. Besides, the system may respond too slowly to dynamic configuration changing.

### **Deterministic vs Probabilistic Algorithms**

The self-stabilizing algorithms can also be divided into two classes: *deterministic* and *probabilistic* (randomized) algorithms. This criterion has nothing to do with the underlying distributed system but concerns with the algorithm design strategy. Randomization is normally used to break the symmetry in anonymous networks. Many randomized algorithms succeed with probability  $1 - \epsilon$ ,  $\epsilon > 0$  (the success is not certain). Besides, the random number generators used are actually pseudo random number generators and some undesirable correlations may appear between neighboring nodes.

### 1.3 Sensor Network

Currently, there has been a lot of interests in the theory and application of sensor networks and due to varied application domains, the sensor networks are also widely different in terms of capabilities and inherent infrastructure support. A very detailed survey of different kinds of sensor networks and their applications can be found in [ASC02]. Our purpose in this chapter is to use some kind of an abstract model of sensor networks [Her03b, KA03] in order to best suit the standard paradigms of designing self-stabilizing distributed algorithms for ad hoc networks and study some of the known protocols under the so called sensornet model. For our purpose, the sensor network is built from a set of (sensor) nodes with same computational and communication capabilities. There is no external backbone network or message repeater facility. Each node  $p$  can communicate with a subset of nodes, called neighbors of  $p$ , determined by the range of the radio signal of  $p$ . Each node uses the same radio frequency. Each node cannot send and receive concurrently, i.e., the communication channels are half-duplex. Nodes do not have collision detection hardware. The network is asynchronous.

#### 1.3.1 Unbounded Asynchronous Unison [Her03b]

To coordinate the need for phase synchronization in distributed system, the problem of asynchronous unison becomes important. To present our algorithm on asynchronous unison, we define the following identifiers and concepts. Let  $G = (V, E)$  be a finite, undirected and connected graph. Let  $v_p$  be the clock of node  $p$  in the Unison system. Let  $K$  be the upper bound of the unison algorithm. Let  $N$  be a positive integer which defines the *behind* relationship in the following subsection. A *state* of the system is defined by a value for every clock variable and a value for every local variable in each node program in the system. A *system transition* is a pair  $(r, s)$  of system states such that there is a rule that can be fired, the system starting from  $r$ , yields state  $s$ . A *system computation* is a maximal sequence of system states such that every pair of successive states is a transition. The legitimacy prediction for an unbounded asynchronous unison system is:  $L \equiv (\forall p, q : (p, q) \in E : |(v_p - v_q)| \leq 1)$ . The system stabilizes to satisfy  $L$  and every node  $p$  executes  $v_p := v_p + 1$  infinitely often. In [Her03b], the following algorithm is proposed and its

correctness is proved.

**Algorithm A-Unison**

**Rule 1.** if  $\forall q : (p, q) \in E, v_p \leq \boxtimes_p v_q$   
 then  $v_p := v_p + 1$ ;

In the next section, we present the algorithm for the bounded version of asynchronous unison in sensor network.

### 1.3.2 Bounded Unison

The legitimacy prediction for a bounded asynchronous unison system is:  $L \equiv (\forall p, q : (p, q) \in E : |(v_p - v_q) \bmod K| \leq 1)$ . Please note that  $v_p - v_q$  can be viewed as either a positive integer or a negative integer in modulo  $K$ . We always use the one with a smaller absolute value. The correctness of a unison algorithm satisfies the following three properties.

**Liveness:** Every computation of the system is infinite.

**Progress:** Every infinite computation of the system has an infinite suffix where every clock variable  $v_p$  is updated infinitely often and only by executing the assignment statement  $v_p = v_p + 1 \bmod K$ .

**Asynchronous Unison:** Every infinite computation of the system has an infinite suffix where each state satisfies legitimacy prediction  $L$ . For the bounded algorithm, the conduction is: for any neighboring nodes  $p$  and  $q$ ,  $v_p = v_q \cup v_p = v_q + 1 \bmod K \cup v_q = v_p + 1 \bmod K$ .

To present the algorithm of bounded unison, we introduce the following two relationships. Relationship *behind* is an extension to the greater equal than in the unbounded unison. The two relationships assist to achieve the self-stabilization of bounded unison in sensor network.

Relation *behind*  $\heartsuit$ :

Let variable  $x$  and  $y$  range over  $0 \dots K - 1$ ,  $x \heartsuit y = ((y - x) \bmod K \leq N)$ .

Relation *far*  $\diamond$ :

Let variable  $x$  and  $y$  range over  $0 \cdots K - 1$ ,  $x \diamond y = \neg(x \heartsuit y) \cap \neg(y \heartsuit x)$ .

In a sensor net model, the condition  $L$  becomes  $C \equiv (\forall p, q : (p, q) \in E : \boxtimes_p v_q \heartsuit v_q)$ .

The following algorithm which achieves an asynchronous unison system executes at every node  $p$  in the network.

**Algorithm B-Unison**

**Rule 1.** **if**  $\forall q : (p, q) \in E, v_p \heartsuit \boxtimes_p v_q$

**then**  $v_p := v_p + 1 \bmod K$ ;

**Rule 2.** **if**  $\exists q : (p, q) \in E, v_p \diamond \boxtimes_p v_q \cap v_p > v_q$

**then**  $v_p := 0$ .

**Lemma 1**  $C$  is an invariant of  $L$  of any execution.

**Proof :** If  $\boxtimes_p v_q$  is an estimation that is behind the value of  $v_q$ , then rule 1 may block  $p$ 's progress, however, whenever  $\boxtimes_p v_q$  is updated, the result of incrementing  $v_p$  satisfies  $L$ . Then, the same argument applies to the neighbors of  $p$ . If the cache update message for the increment is not correctly received by a neighbor  $q$ , rule 1 may block that neighbor's progress.  $\square$

The clock variables in unison systems are time-driven. They increment infinitely even if no other significant events occur in the system. Therefore every computation of the system is infinite. We show the proposed algorithms in asynchronous sensor network have an infinite suffix where each state satisfies condition  $C$ .

**Lemma 2** With probability 1, every execution eventually satisfies  $L \cap C$ .

**Proof :** If a node  $p$  never increments  $v_p$ , the system will eventually be deadlocked and cache coherent. This contradicts the self-stabilizing behavior of the algorithm.

In order for  $p$  to infinitely increment  $v_p$  in modulo  $K$ , it must forever correctly receive messages from each neighbor  $q$  containing values of  $v_q$ . In turn, each neighbor  $q$  must correctly receive up-

dated clock values from all its neighbors; by an inductive argument it follows that all nodes increment their clocks infinitely often. Therefore, from rule 1 and lemma 1, every execution eventually satisfies  $C \cap L$ ,  $\square$

## 1.4 Maximal Independent Set

Let  $G = (V, E)$  be a finite, undirected and connected graph. Let  $s_p$  be a variable of node  $p$  in the network.  $s_p$  indicates the membership of the node in the maximal independent set. Authors in [Her03a] proposed an id based algorithm for leaders via maximal independent set. We propose here an algorithm that does not use ids.

If  $s_p = 1 \cap (\forall q : (p, q) \in E, s_q = 0)$ , refer to as  $L_1$ , we say that the node  $p$  is independent. If  $s_p = 0 \cap (\exists q : (p, q) \in E, s_q = 1)$ , refer to as  $L_2$ , we say that the node  $p$  is dominated.

In a legitimate result, each node in the network should satisfy  $L_1 \cap L_2$ . In a sensor network, condition  $L_1$  becomes  $s_p = 0 \cap (\exists q : (p, q) \in E, \boxtimes_p s_q = 1)$ , refer to as  $C_1$ . Condition  $L_2$  becomes  $s_p = 0 \cap (\exists q : (p, q) \in E, \boxtimes_p s_q = 1)$ , refer to as  $C_2$ . For this algorithm, the weakened cache coherent model refers to that the cached values may be stale.

The following algorithm which achieves a maximal independent set executes at every node  $p$  in the network.

### Algorithm M-Independent-Set

**Rule 1.** if  $\forall q : (p, q) \in E, s_p = 0 \cap \boxtimes_p s_q = 0$

**then**  $v_s := 1$ ;

**Rule 2.** if  $\exists q : (p, q) \in E, s_p = 1 \cap \boxtimes_p s_q = 1$

**then**  $s_p := 0$ .

Without the knowledge of coherency of a cache entry, the system may come to a stable state but later may reconfigure due to a recognized stale cache entry. In paper [] Self-stabilizing Algorithms for Minimal Dominating sets and Maximal Independent Sets, the following lemma is supplied. We include it here without further detail.

**Lemma 3** *The algorithm achieves  $C_1 \cap C_2$  upon stabilization.*

**Lemma 4** *With probability 1,  $C_1 \cap C_2$  is an invariant of  $L_1 \cap L_2$  of any execution.*

The following algorithm appeared in [Her03a]. Each node in the network have a Boolean variable  $L$ . In an initial state, the value of  $L$  is arbitrary.

**Algorithm Leaders-M-Independent-Set**

**Rule 1.** **if**  $\forall q \in N(p) : p > q$   
**then**  $L_p = true$ ;

**Rule 2.** **if**  $\exists q \in N(p) : \boxtimes_p L_q \wedge q < p$   
**then**  $L_p = false$ ;

**Rule 3.** **if**  $(\exists q \in N(p) : q < p) \wedge (\forall q \in N(p), q > p : \neg \boxtimes_p L_q)$   
**then**  $L_p = true$ ;

The algorithm does not use randomization explicitly. However, its convergence is probabilistic because the underlying model of communication uses random delay. The paper shows that with probability 1, the algorithm converges to a maximal independent set with convergence time  $O(1)$ .

**Proof :** The probability of a node in sensor network with eventual coherent cache is 1. Let  $p$  be one of the nodes that contains stale cache values of  $\boxtimes_p s_q$ . Once the node received a correctly updated value of  $s_q$ , it will trigger a round of updating in the network. Subsequently, each such error will trigger a round of updating. Let  $\tau$  be the probability of one or more nodes failed to receive a message. If  $\tau$  is less than 1, the probability of the system infinitely executing without achieving  $L_1 \cap L_2$  is 0. Upon stabilization,  $C_1 \cap C_2$  is equivalent to  $L_1 \cap L_2$ .  $\square$

## 1.5 Minimal Domination

Using the notation from last section, a legitimate state of the network will satisfy  $L_2$ . A weakened condition in sensor network is that every node satisfies  $C_2$ .

In [HHJS03], the following lemma is proved.

**Lemma 5** *A set  $S$  is a minimal dominating set iff its dominating and every  $u \in S$  has a private neighbor.*

We use the variable  $d_p$  to denote the pointer at node  $p$  used in the algorithm. The value of  $d_p$  is the identifier for a node, such as  $q$ . The pointer variable  $d_p$  is used to indicate the private neighbor relationship.

The following algorithm which finds a minimal dominating set executes at every node  $p$  in the network. Let  $N(p)$  denote the open neighborhood of node  $p$ .

**Algorithm M-Dominating-Set**

**Rule 1.** if  $v_p = 0 \wedge \forall q \in N(p) : \boxtimes_p v_q = 0$   
     **then**  $v_p = 1$ ;

**Rule 2.** if  $v_p = 1 \wedge \forall q \in N(p) : \boxtimes_p d_q \neq p \wedge \exists k \in N(p) :$   
 $\boxtimes_p v_k = 1$   
     **then**  $v_p = 0$ ;

**Rule 3.** if  $v_p = 1 \wedge d_p \neq NULL$   
     **then**  $d_p = NULL$ ;

**Rule 4.** if  $v_p = 0 \wedge \exists! q \in N(p) : \boxtimes_p v_q = 1 \wedge d_p \neq q$   
     **then**  $d_p = q$ ;

**Rule 5.** if  $v_p = 0 \wedge \exists$  **more than one**  $q \in N(p) : \boxtimes_p v_q =$   
 $1 \wedge d_p \neq NULL$   
     **then**  $d_p = NULL$ ;

**Lemma 6** *With probability 1,  $C_2$  is an invariant of  $L_2$  of any execution and is achieved by every execution.*

**Proof :** The probability of a node in sensor network with eventual coherent cache is 1. Let  $p$  be one of the nodes that contains stale cache values of  $\boxtimes_p s_q$ . Once the node received a correctly updated value of  $s_q$ , it will trigger a round of updating in the network. Subsequently, each such error will trigger a round of updating. Let  $\tau$  be the probability of one or more nodes failed to receive a message. If  $\tau$  is less than 1, the probability of the system infinitely executing without achieving  $C_2 \cap L_2$  is 0.  $\square$

## 1.6 Neighborhood Identification

Because the sensor network is ad hoc, therefore there is a need to configure the neighborhood knowledge of each node. At a node  $p$ , we can represent  $N(p)$  by a list of identifiers learned by receiving messages from neighboring nodes. Let  $L$  be a list of nodes recording the close neighborhood of a node. Note that we do not include an aging mechanism for this algorithm because the sensor network model is not mobile. The following algorithm achieves neighborhood identification.

### Algorithm Neighborhood

**Rule 1.** if receive message from  $q$ , never receive any message before

**then**  $\boxtimes_p L := \{q\}$ ;

**Rule 2.** if receive message from  $q$

**then**  $\boxtimes_p L := \boxtimes_p L + q$ ;

**Lemma 7** *The algorithm recognizes the neighborhood of each node.*

**Proof :** With probability 1, a node  $p$  receives all the messages from its neighbors. Therefore, each node  $p$  can configure the neighborhood list correctly.  $\square$

## 1.7 Neighborhood Unique Naming

Let  $N3_p$  denote the distance three neighborhood of node  $p$ . The Algorithm finds a unique color for each node that is distinct within  $N3_p$  neighborhood. Define namespace  $\Delta = [\delta^t]$  for some  $t > 3$ . Let  $Id$  be the unique naming,  $Id \in \{0 \cdots \Delta\}$ . Let set  $Cids_p = \{\boxtimes Id_q | q \in N3_p \setminus \{p\}\}$ .

The following algorithm appeared in [Her03a]; we include it here without proof.

<p><b>Algorithm Leaders-M-Independent-Set</b></p> <p><b>Rule 1.</b> if <math>Id_p \in Cids_p</math></p> <p>    <b>then</b> <math>Id_p := \mathbf{random}(\Delta \setminus Cids_p)</math>;</p>
---

## 1.8 Acknowledgement

The work reported in this chapter was supported by an NSF award ANI #0218495.

## References

- [Ang80] D. Angluin. Local and global properties in networks of processors. In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 82–93, 1980.
- [ASC02] I. F. Akyildiz, W Su, and E. Cayirci. A survey on sensor networks. *IEEE Communication Magazine*, pages 102–115, August 2002.
- [BHG87] P.A Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems*. Addison-Wesley, 1987.
- [DGRS03] A. K. Datta, M. Gradinariu, M. Raynal, and G. Simon. Anonymous publish/subscribe in P2P Networks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.
- [Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
- [GHJS03] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing distributed algorithm for strong matching in a system graph. In *Proceedings of HiPC 2003*, volume LNCS 2913, pages 66–73, Springer Verlag, 2003.
- [Gho01] S. Ghosh. Cooperating mobile agents and stabilization. In *Proceedings of WSS-2001*, volume LNCS 2194, pages 1–18, Springer-Verlag, 2001.
- [Gou95] MG Gouda. The triumph and tribulation of system stabilization. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings*, Springer-Verlag LNCS:972, pages 1–18, 1995.

- [GRS03] M. Gardinariu, M. Raynal, and G. Simon. Looking for a common view for mobile worlds. In *Proceedings of the Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, 2003.
- [Her03a] T. Herman. A distributed TDMA slot assignment algorithm for wireless sensor networks. Technical Report LRI 1370, Universit Paris Sud, September 2003.
- [Her03b] T. Herman. Models of self stabilization and sensor networks. In *Proceedings of IWDC 2003*, volume LNCS 2918, pages 205–214, Springer Verlag, 2003.
- [HHJS03] S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers and Mathematics with Applications*, 46:805–811, 2003.
- [Hua93] S. T. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.
- [KA03] S. K. Kulkarni and U. Arumugam. Collision free communication in sensor networks. In *Proceedings of SSS 2003*, volume LNCS 2704, Springer Verlag, 2003.
- [Kes88] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Inf. Processing Letters*, 29(2):39–42, 1988.
- [ST96] Sartaj Sahni and Venkat Thanvantri. Parallel computin: Performance metrics and models. Technical Report TR-008, University of Florida, Gainesville, 1996.