

# Self-stabilizing Network Algorithms : A Survey

Tridib Mukherjee  
ASU ID : 993-76-1704

April 21, 2005

## Abstract

Self-stabilization for distributed system was first proposed by Dijkstra in 1974. Till then it has received a huge amount of attention in the distributed system community. The attractive way in which it does fault-tolerance by taking local actions in the distributed nodes made it one of the most popular ways of fault-tolerance.

Many self-stabilizing algorithms had been developed for various different types of distributed networks such as token ring systems, spanning trees in networks etc. In this term-paper, a survey on self-stabilizing network algorithms has been done with its implications in the corresponding systems.

## 1 Introduction

One of the most wanted properties of distributed systems is fault tolerance. This can be achieved, in general, by two different approaches: pessimistic and optimistic. In the former, we deal with robust algorithms protected against any possible (i.e. the most pessimistic) or admissible set of failures. In the latter case, we use selfstabilizing algorithms, which after any failure guarantee to automatically reach a legal state in a finite time. In general, self-stabilizing algorithms need not know anything about the failure, its type, duration, scope nor even whether it actually happened or not. Self-stabilization is an attractive technique of doing fault-tolerance in distributed systems by taking only local actions in the distributed nodes.

A self- stabilizing algorithm does not require correct initialization, can recover from any transient failure of arbitrary types occurring at any time and is insensitive to dynamic topology reconfiguration. This makes self-stabilization an interesting solution for a large number of applications including distributed reset problems, routing and communication protocols , clock synchronization , graph theory , and others. Most of the work in the literature for self-stabilizing network protocols has focused on maintaining spanning trees in a network in a self-stabilizing manner. In this term paper, we do a survey on the self-stabilizing network algorithms.

Self Stabilization was proposed in a famous paper by Dijkstra in 1974. He defined a system as self-stabilizing when regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps. A system which is not self-stabilizing may stay in an illegitimate state forever. Dijkstra's notion of self-stabilization, which originally had a very narrow scope of application, is proving to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed systems. The main idea is to come up with a distributed system that stabilizes automatically from any arbitrary initial state. Thus, the notion of initial state is not there in a self stabilizing distributed system. However, Self Stabilization is used as a fault-tolerance scheme in distributed systems as it converges to a legitimate state from any faulty state. This property of Self-Stabilization is known as Convergence. Another property of this scheme is that the system does not go to a faulty state unless a fault occurs. Therefore, once in a legitimate state, the system remains in a legitimate state. This property is called Closure.

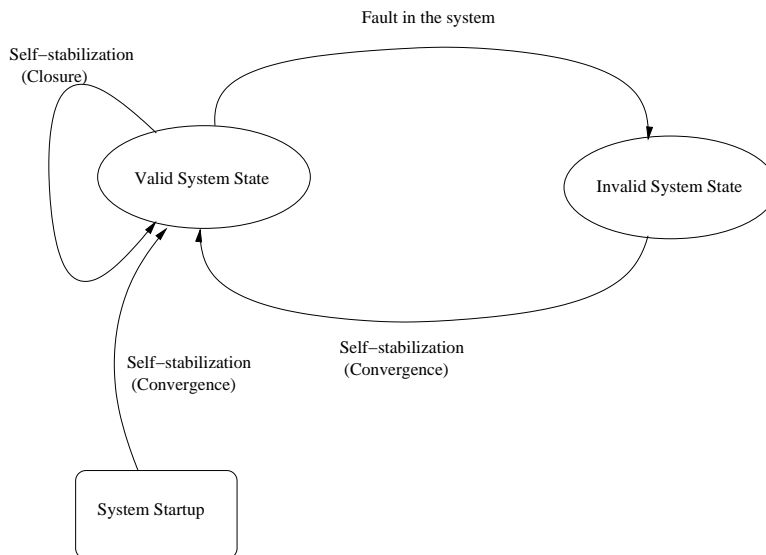


Figure 1: State diagram of a system running self-stabilization

Figure 1 shows a state diagram for the self-stabilizing systems. At the time of system start-up, the self-stabilizing system is guaranteed to come to a valid system state from any arbitrary initial state. Although, the start up can be merged with the invalid state, but it is also possible that the initial state is not invalid. Thus, the start up of the system is shown separately.

Section 2 discusses the first self-stabilizing paper for distributed systems by Dijkstra, along with another algorithm for quite similar settings by George Varghese. Section 3 gives the survey on the various algorithms for self-stabilizing network algorithms. Then, self-stabilizing protocols are discussed for other modern networks like mobile ad-hoc networks. An optimization scheme to improve the self-stabilizing network algorithms is discussed in section 5. We conclude by discussing the issues of self-stabilization in the novel regime of sensor networks.

## 2 Dijkstra's Mutual Exclusion in Token Rings

One of the clearest and most demonstrative examples of self-stabilizing algorithms is, already a classic, mutual exclusion in a ring of  $n$  processes, published by Dijkstra in 1974 [1]. It requires the number  $k$  of local states to be not less than the number of processes.

Dijkstra introduced the notion of a privilege. A privilege authorizes a given process to make a move (i.e. change its local state, enter the critical section). The legal (global) system state must satisfy the following properties:

1. There must be at least one privilege in the system.
2. During an infinite time every process should be able to receive a privilege an infinite number of times.

There is one exceptional process  $P_0$  in the ring. It follows different rules than the other  $n - 1$  ( $P_1$  to  $P_{n-1}$ ) processes which are equal and behave uniformly. The current local state of a given process  $P_i$  is represented by a state variable  $l_i$ . The algorithm assumes any model of communication that lets every process in the ring know the value of the state variable of its left neighbor ( $l_{i-1}$  for  $1 \leq i \leq n - 1$ , and  $l_{n-1}$  for  $P_0$ ).

Here is Dijkstra's algorithm:

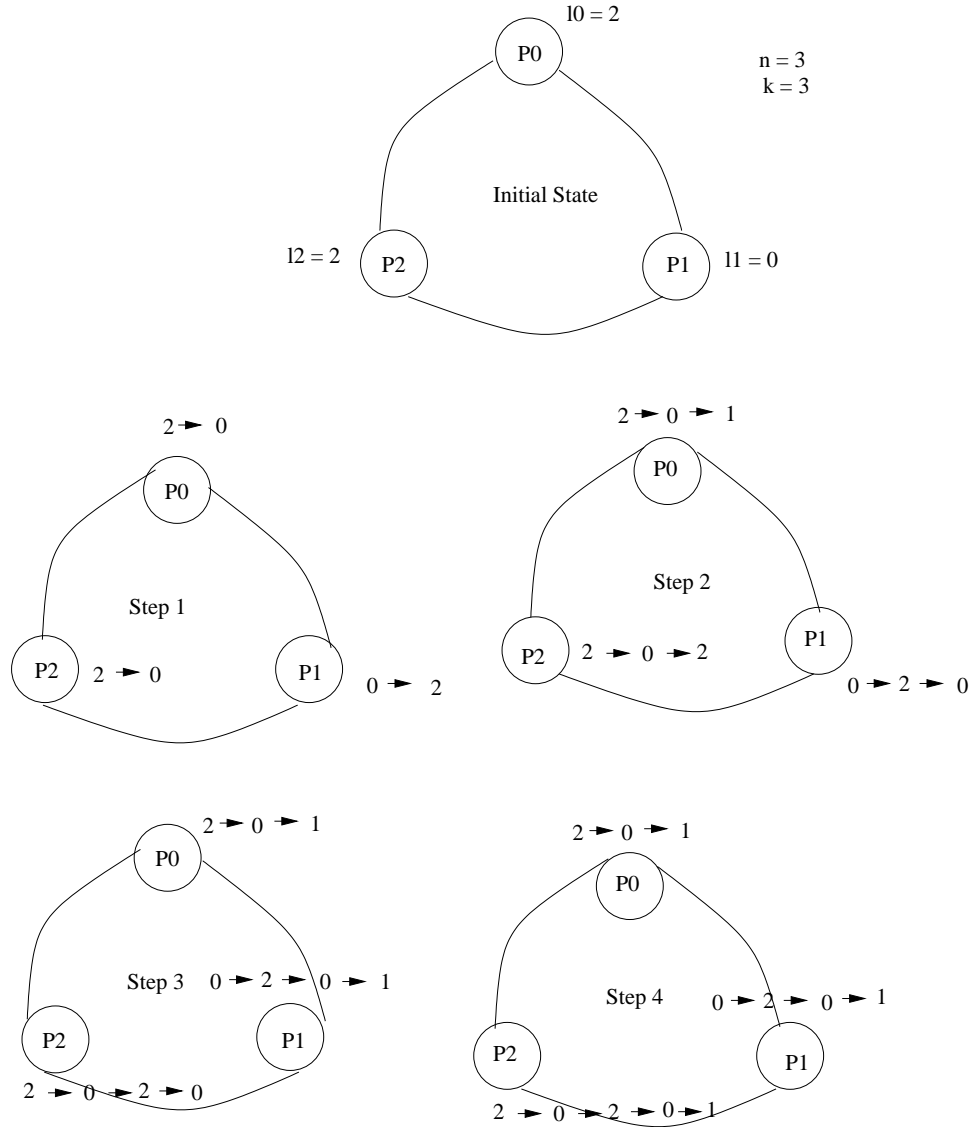


Figure 2: Example run of Dijkstra's self-stabilizing mutual exclusion algorithm

for  $P_0$  : if  $l_0 = l_{n-1}$  then  $l_0 := (l_0 + 1) \bmod k$  and  $P_0$  has a privilege;  
for  $P_i$  : if  $l_i \neq l_{i-1}$  then  $l_i := l_{i-1}$  and  $P_i$  has a privilege;  $1 \leq i \leq n - 1$

Figure 2 shows an exemplary run of this algorithm in a ring of 3 processes, where  $k=3$ . Note that because of the assumed initial state, all three processes get a privilege (process  $P_0$  satisfies the condition  $l_0 = l_{n-1}$ , process  $P_1$ :  $l_1 \neq l_0$ , and process  $P_2$ :  $l_2 \neq l_1$ ) thus all three enter their critical sections and change their state in step 1. This global state is obviously not legal from the mutual exclusion point of view. Then, once again, all of them receive a privilege leading to step 2 (another illegal global state). Now, condition  $l_0 = l_{n-1}$  is violated, process  $P_0$  cannot get into its critical section, but  $P_1$  and  $P_2$  still can (this is why only  $P_1$  and  $P_2$  change their state in step 3; the new state is still illegal). In the configuration after step 3 only  $P_2$  has a privilege, the system has converged to a set of legal states. From now on, only one process will have a privilege. The privilege will circulate clockwise in the ring, e.g. in step 4: from  $P_1$  to  $P_2$ , next from  $P_2$  to  $P_0$ , etc.

After this basic paper by Dijkstra, there has been a lot of work on top of this. One example is that of Counter

flushing in the network done by George Varghese [21].

A counter is periodically broadcast by a leader in the network. The leader has the privilege of incrementing the counter in each broadcast. Self-stabilization guarantees that the counter is delivered to all the nodes without running into any deadlock or livelock. Varghese gave Counter flushing algorithms for token ring settings and tree settings based on Propagation of information with feedback (PIF). For the tree based approach, the root node acts as the leader for counter flushing. They also broadened the scope of counter flushing to general graphs and illustrated the idea for network reset protocols.

### 3 Self-stabilizing Network Algorithms

Fault-tolerance is very important in modern day networks. Due to the increase in the network sizes, the probability of a fault in the network has also increased. Moreover, with the inclination towards the wireless networks, various kinds of faults can be encountered due to error prone medium, mobility of the network nodes, limitation of the battery power to name a few. Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. So, Fault Tolerance makes the network system more dependable. Faults can be of two basic types, *transient* and *permanent*. While it may seem that permanent faults are more severe, from an engineering perspective they are much easier to diagnose and handle. The intermittent transient faults that recur often unpredictably are the most problematic. Self-Stabilization is a proactive technique that is used for fault tolerance in distributed networks as illustrated in the previous sections.

Most of the research in self-stabilizing network algorithms focused on the maintenance of the spanning trees in a distributed network. A spanning tree in the network is often a prerequisite for more involved network protocols like routing or token circulation. It can also increase the efficiency of network protocols. Take for example the problem of broadcasting messages in the network. There are algorithms which flood the network, i.e., the broadcast message is sent to all neighbors. Consequently, the message crosses all communication links before the protocol has finished. However, if a spanning tree of the network is available, the message only needs to be sent to all those nodes which are neighbors in the spanning tree. Instead of crossing all links, it just crosses  $n-1$  links (which is significantly less than the total number of links in the network and thus a spanning tree can considerably reduce the message complexity of the broadcast algorithm).

These algorithms solve a basic problem in a very robust way and can be used as building blocks in fault-tolerant and dynamic applications. The first self-stabilizing constructions algorithms were published in the beginning of the 1990s.

#### 3.1 Self-stabilizing BFS spanning tree construction Algorithm by Dolev, Israeli and Moran

Dolev, Israeli and Moran [3] [4] in 1990 proposed 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 of 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 root process. After sufficient activations of the root, it

has written 0 values into all of its output variables. These values will not change anymore. Without a distinguished root process the distance values in all nodes would grow without bound. More specifically, after reading all neighbor's 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  $O(\text{section})$  update cycles the entire tree will have stabilized. The above algorithm is used by Dolev as the basis for a topology update algorithm in dynamic networks. Based on the same algorithmic idea, Collin and Dolev [5] present a semi-uniform spanning-tree algorithm under a central daemon and read/write atomicity that constructs a DFS tree (instead of a BFS tree). A similar algorithm which also constructs a DFS tree but uses composite atomicity was published by Herman in his Phd thesis three years earlier. In this algorithm, the outgoing links at every process are ordered, and the DFS tree is defined as the tree resulting from a DFS graph traversal always selecting the smallest outgoing edge. Instead of writing its current level into the output registers, it writes a representation of its current estimate of the path (the sequence of outgoing link identifiers) to the root. The root repeatedly writes the empty path to its output registers. If a node has  $k$  neighbors, there are  $k$  alternative paths to choose from. From these, the node chooses the path which is minimal according to a lexicographic order which prefers smaller link identifiers, and so a node does not choose the shortest path to the root but along the smallest link identifiers. The authors remark that this is the same principle as used in the algorithm of Dolev, Israeli and Moran [3]. The memory requirements for the DFS algorithm however are  $O(n \log K)$  bits where  $K$  is an upper bound on the maximum degree of a node. The time complexity is  $O(\text{section} n K)$  rounds.

### 3.2 BFS spanning-tree algorithm by Afek, Kutten and Yung

Afek, Kutten and Yung [7] presented a 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.

### 3.3 Self-stabilizing BFS spanning-tree algorithm by Arora and Gouda

Also in 1990, Arora and Gouda [10] [11] published a self-stabilizing BFS spanning tree algorithm for the composite atomicity model under a central daemon. Similar to Afek, Kutten and Yung, they also assume unique identifiers and the node with maximum identifier eventually acts as the root of the system. In contrast to Afek, Kutten and Yung, the algorithm needs a bound  $N$  on the number  $n$  of nodes in the network to work correctly. The bound  $N$  is necessary because the algorithm uses a different technique to detect and remove cycles. Again, every node maintains variables for distance, parent and root identifier. Periodically, every node compares its own distance and root identifier setting

with the values stores in the node pointed to by the parent variable. In the finished spanning tree, the root identifiers should be the same and the distance should be the distance of the parent incremented by 1. If this is not the case, the root identifier is copied from the parent and the distance is set to the parent's distance plus 1. If there is a cycle in the tree (for example due to improper initialization), the distance values are incremented along this cycle without bound. Hence, a cycle is detected when the distance value supercedes the bound  $N$ . The first node to detect this makes itself the root of a new tree. A node also continuously monitors the root identifier and distance settings of its neighbors. If a neighbor has a larger root identifier or the same identifier with smaller distance, the node adjusts its values accordingly. Knowledge of the bound  $N$  allows the algorithm of Arora and Gouda [11] to be simpler than the one by Afek, Kutten and Yung [7] but the stabilization time is  $O(N^2)$ , which can be much larger than  $O(n^2)$ . In dynamic networks where network nodes may go down, a stabilization time in the order of the actual number of nodes is preferable.

### 3.4 The Algorithms by Huang et al.

The same idea for cycle breaking (through bumping up the distance counter) was presented in 1991 by Chen, Yu and Huang [12]. In this paper, they present a self-stabilizing spanning tree algorithm for semi-uniform systems with composite atomicity. The fact that there is a distinguished root makes the algorithm even simpler than the one by Arora and Gouda [11]. However, the algorithm does not necessarily stabilize to a BFS tree since the choice of a new parent after breaking a cycle is non-deterministic (governed by the scheduler). This was adapted in a later paper by Huang and Chen [13] to yield an algorithm which constructs a BFS tree using knowledge of the size  $n$  of the network. Interestingly, Huang and Chen [13] see the contribution of the latter algorithm in the technique to prove stabilization, not in the algorithm itself since the one by Dolev, Israeli and Moran [3] achieves the same goal but assumes read/write atomicity instead of composite atomicity. Four years after Chen, Yu and Huang's paper [12], the algorithm was re-invented (with slight modifications) by Antonoiu and Srimani [14]. In this paper too, the authors claim that their proof technique is as important a contribution as the algorithm itself.

### 3.5 The Algorithm by Afek and Bremner

Afek and Bremner [8] [9] revisit the problem of self-stabilizing spanning-tree construction. They give an algorithm for systems with unidirectional, bounded capacity message passing links. They assume unique identifiers and give adoptions of the algorithm for the synchronous and the asynchronous network case. The network node with the minimum identifier eventually plays the role of the root in the spanning tree. The algorithm exploits a new design idea called power supply which enables the algorithm to have unique features. For example, the algorithm stabilizes in  $O(n)$  rounds without any knowledge about  $n$ . The power supply method exploits the fact that self-stabilizing algorithms must continuously check their own state. In the algorithm, nodes which are part of some spanning tree expect to receive power from the root of the tree (power means a continuous flow of certain messages, one per round). The idea of the algorithm is that only legal roots may be the source of power and that nodes attached to fake roots eventually fail to receive power and subsequently make themselves the root of a new tree. Whenever a node receives power from a neighbor with a smaller identifier, it attaches itself to its tree. In the asynchronous case, the power supply idea is implemented using using different types of messages: weak messages are exchanged periodically between the nodes to synchronize their state, while strong messages carry power. Afek and Bremner give a generic power supply algorithm which can be instantiated to a leader election algorithm, or an algorithm to construct DFS or BFS spanning trees.

### 3.6 Other Related Work

In all of the above algorithms, processes have to maintain a variable measuring the distance from the root of the tree which must be communicated to the neighbors. This means that communication variables must have at least  $O(\log n)$  bits. Work by Awerbuch and Ostrovsky [15] reduce this requirement to  $O(\log^* n)$  bits per edge. Awerbuch, Patt-Shamir and Varghese [16] give an algorithm based on unique identifiers within the paradigm of local checking and correction. Aggarwal and Kutten [18] [19] give a spanning-tree algorithm for message passing environments which uses a clever data structure to allow stabilization with  $O(\text{section})$  time without any knowledge of network diameter or number of nodes. The basic version of the algorithm was designed for anonymous networks and uses randomization to create unique identifiers. Awerbuch et al. [17] present another time-optimal spanning-tree algorithm which is based on unique identifiers, but it requires knowledge of a bound on the network diameter. Similarly, Dolev, Israeli and Moran [6] present an algorithm for anonymous networks which stabilizes in  $O(\text{section})$  rounds but requires knowledge of a bound  $N$  on network size. Other authors have investigated other flavors of spanning-tree construction (e.g., minimum diameter spanning tree and minimum spanning tree in a graph with weighted edges).

## 4 Self-stabilization in mobile ad hoc networks(MANETs)

In mobile ad hoc networks, adaptivity is essential as the topology of the network is dynamic due to mobility of network nodes and energy saving of the nodes. Self-stabilization paradigm is thus suitable to adapt to the transient faults caused due to the topological changes. First approach in this regard was given by Gupta and Srimani in [2]. They presented self-stabilizing Shortest Path Spanning Tree (SS-SPST) and self-stabilizing Minimum Spanning Tree (SS-MST) protocols for multicasting operation in the mobile ad-hoc networks. Each node maintains information about its parent in the tree and its current level. These information are broadcast periodically among the neighbor nodes. Thus each node takes local actions based on local information (i.e information in itself and its neighbors) to construct SPST. For MST, each node maintains one additional information of the minimum route from the root to that node. Both the algorithms construct multicast tree in a MANET in three logical steps. Firstly, it constructs a shortest-path spanning tree (or minimum spanning tree) of the mobile network graph in presence of topology change due to node mobility. Secondly, the algorithms root the source node with respect to the stabilized multicast tree. Finally, they prune the nodes from the constructed tree that are not involved in multicasting. The purpose of the algorithms is to bring the system back to the stable or legitimate state whenever it enters an illegitimate state due to perturbation. Each node looks at the states of its neighbors and checks the local satisfiability of the global legitimate state; if the node is not locally legitimate, it triggers self-stabilization routine and tries to be in the legitimate state. The upper bound on the number of rounds needed by the entire network to stabilize starting from an arbitrary illegitimate state is given by  $\mathcal{D} \cdot \lceil \frac{m_2}{m_1} \rceil + 1$  where  $m_1$  and  $m_2$  denote respectively the minimum and maximum edge cost in the system graph and let  $\mathcal{D}$  be the diameter of the underlying network in terms of the number of edges traversed. Both the algorithms have the same performance. However, SS-MST will always guarantee a tree with minimum cost but SS-SPST does not do that.

## 5 Discussion

In our survey for self-stabilizing network algorithms, we have discussed couple of algorithms for the token ring settings and rest for the spanning tree settings. An example run of the Dijkstra's mutual exclusion algorithm has been shown in section 2. In this section, an example run is shown for the spanning trees. Figure 3 is a valid spanning tree where R is the root node. But, some occurrence of fault takes the system as in Figure 4. This state is invalid as the root

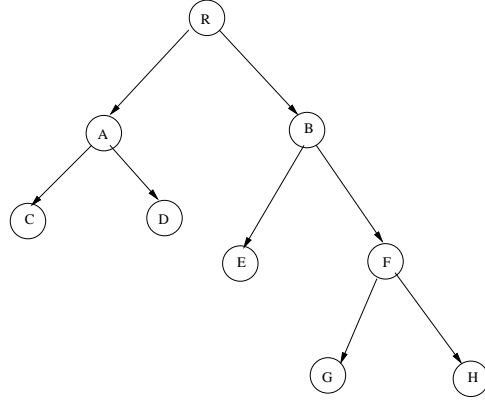


Figure 3: Original Spanning Tree (Legitimate State)

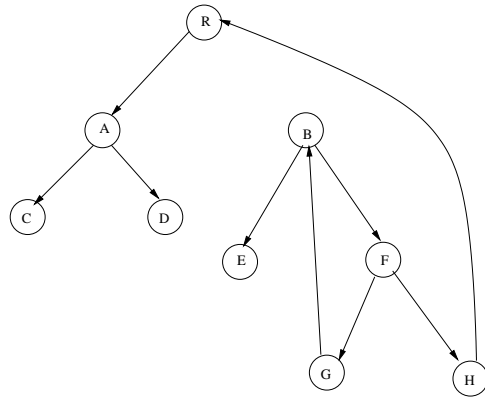


Figure 4: Faults in the original tree creates this illegitimate state

node R has a parent and also there is a cycle between nodes B, F and G. If root node R takes action before any other node then the system gets in the state as in Figure 5. However, this is not the valid state as the cycle is still there. When node B takes action, the tree goes to the valid state as in Figure 6. Note that this tree is the same as the initial valid tree of Figure 3. The example run is given in general for any self-stabilizing spanning tree algorithm.

In the above discussion of self-stabilizing network algorithms, when the system is in the process of stabilization it is possible that some of the network nodes are not reachable as the spanning tree may not be constructed fully until it is stabilized. For example, in Figure 5, node B is not reachable from R. Thus, there is a possibility of service unavailability during the period when the system is in the process of stabilization. This unavailability may increase if the system takes greater time to stabilize. Minimizing this stabilization latency increases the availability of the service.

Intuitively it can be said that if the tree information of a node is changed it should be enough to take corrective measures only in the faulty node instead of several other nodes. In Figure 7 we consider SPST that has already been established. Here, both C and F are in the range of D initially. When C goes out of the range of node D, it should be enough to change the parent information of node D. However, if node E acts earlier then the fault propagates down the tree. This takes unnecessarily long time and energy to stabilize the system. Proper scheduling of node actions can dissolve the problem but scheduling among distributed nodes can be very expensive when many distributed nodes gets involved and co-ordinate with each other. In [22] and [23], the above issue of optimizing the stabilization latency is discussed in general. The authors have produced fault-containment over self-stabilization to

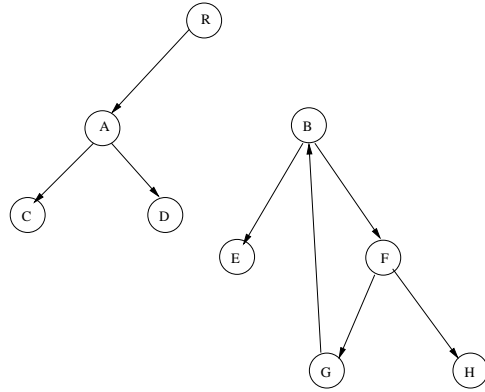


Figure 5: Fault in the Root node is eliminated

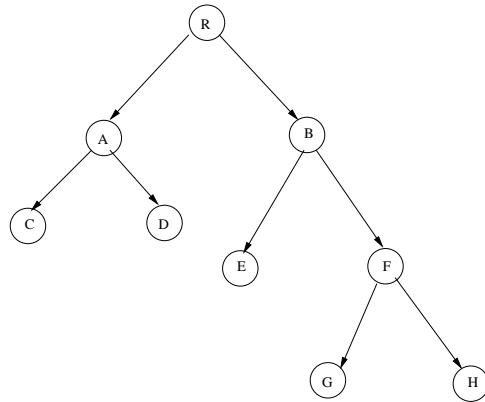


Figure 6: Fault in node B is also eliminated: the tree gets stabilized in the legitimate(fault-free) state

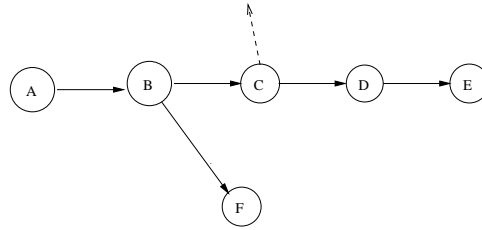


Figure 7: Single Transient Fault

prevent propagation of limited transient faults. There work has been done over the BFST algorithms of Huan and Chen.

## 6 Conclusion

In this term paper, the self-stabilizing network algorithms are surveyed in a thorough manner. In the existing literature, very few work has been found on self-stabilization in the mobile ad hoc networks and sensor networks. Although, [2] gives a couple of self-stabilizing multicast protocols for mobile ad hoc networks, it does not explore some of the key issues of mobile ad hoc networks such as the energy efficiency. Future work is necessary in creating self-stabilizing algorithms for mobile ad hoc networks in an energy efficient manner so that this hugely adaptive paradigm of the distributed systems becomes feasible for the wireless ad hoc networks.

Moreover, wireless sensor networks is altogether a novel networking regime where all the issues of the mobile ad hoc networks still exists. In addition, sensor network has different kinds of application. For example, apart from maintaining spanning trees for broadcast/multicast, it is important to maintain a spanning tree to do aggregation of data from different sensors. In certain cases, sensor nodes are maintained in a cluster. Reconfiguration of these clusters of sensors can be done using self-stabilization. This reconfiguration is required as sensor nodes may go to sleep in certain times. Moreover, while tracking a moving object in the sensor networks, it is possible that only the nodes that are in the vicinity of the object are required to be kept awake. But due to the mobility of the object, the set of awake nodes also move. These situation can be avoided by incorporating spatio-temporal multicasting in the sensor networks. This special type of multicasting can also be done in a self-stabilizing manner in the future.

In [20] a self-stabilizing model has been given for the sensor network environment. However, in the model Herman talks about the possibility of self-stabilizing cache coherence among the sensor nodes. But the sensor network model assumed in the paper is quite restrictive. Furthermore, besides cache-coherence, there are lots of other practical requirements in the sensor networks. We have already discussed these in the previous paragraph. Addressing these issues in a self-stabilizing manner is a challenge in the future. We need self-stabilization in those situation as it has the internal capability of adaptivity and fault-tolerance. Moreover, as the sensor nodes are likely to be deployed in remote environments, initializing those nodes can become difficult. Self-stabilization eradicates this initialization problem by guaranting the converence to a valid state from any arbitrary initial state as was the case in the algorithms we have discussed in this report.

## References

- [1] E. W. Dijkstra. "Self Stabilizing systems in spite of distributed control". *In Proc. Communications of the ACM*, November 1974.

- [2] P. K. Srimani and S. K. S. Gupta. “Self-Stabilizing Multicast Protocols for Ad Hoc Networks”. *Journal of Parallel and Distributed Computing*, vol. 63, no. 1, pp. 87-96, 2003.
- [3] Shlomi Dolev, Amos Israeli, and Shlomo Moran. “Self-stabilization of dynamic systems assuming only read/write atomicity”. In *Cynthia Dwork, editor, Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 103-118, Quebec City, Quebec, Canada, August 1990. ACM Press.
- [4] Shlomi Dolev, Amos Israeli, and Shlomo Moran. “Self-stabilization of dynamic systems assuming only read/write atomicity”. *Distributed Computing*, 7:316, 1993.
- [5] Zeev Collin and Shlomi Dolev. “Self-stabilizing depth first search”. *Information Processing Letters*, 49:297-301, 1994.
- [6] S Dolev, A Israeli, and S Moran. “Uniform dynamic self-stabilizing leader election”. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424-440, 1997.
- [7] Yehuda Afek, Shay Kutten, and Moti Yung. “Memory-efficient self stabilizing protocols for general networks”. In *Jan van Leeuwen and Nicola Santoro, editors, Distributed Algorithms, 4th International Workshop*, 486:152-8, Italy, 1990.
- [8] Yehuda Afek and Anat Bremler. “Self-stabilizing unidirectional network algorithms by power-supply (extended abstract)”. In *Proceedings of the Eighth Annual ACM- SIAM Symposium on Discrete Algorithms*, pages 111-120, New Orleans, Louisiana, 57 January 1997.
- [9] Yehuda Afek and Anat Bremler. “Self-stabilizing unidirectional network algorithms by power supply”. *Chicago Journal of Theoretical Computer Science*, 1998(3), December 1998.
- [10] Anish Arora and Mohamed Gouda. “Distributed reset”. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, 472:316-333, Springer-Verlag, 1990.
- [11] Anish Arora and Mohamed Gouda. “Distributed reset”. *IEEE Transactions on Computers*, 43(9):1026-1038, September 1994.
- [12] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. “A self-stabilizing algorithm for constructing spanning trees”. *Information Processing Letters*, 39:147-151, 1991.
- [13] Shing Tsaan Huang and Nian Shing Chen. “A self-stabilizing algorithm for constructing breadth-first trees”. *Information Processing Letters*, 41(2):109-117, February 1992.
- [14] Gheorghe Antonoiu and Pradip K. Srimani. “A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph”. *Computers and Mathematics with Applications*, 30:17, 1995.
- [15] Baruch Awerbuch and Rafail Ostrovsky. “Memory-efficient and self-stabilizing network reset”. In *Symposium on Principles of Distributed Computing*, pages 254-263, New York, USA, August 1994. ACM Press.
- [16] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. “Self-stabilization by local checking and correction”. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268-277, 1991.
- [17] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. “Time optimal self-stabilizing synchronization”. *Proceedings of the twenty-fifth annual ACM Symposium on the Theory of Computing*, San Diego, California, May 16-18, 1993, pages 652-661, New York, NY, USA, 1993.

- [18] S. Aggarwal. "Time optimal self-stabilizing spanning tree algorithms". Technical Report MIT-LCS//MIT/LCS/TR-632, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1994.
- [19] S. Aggarwal and S. Kutten. "Time optimal selfstabilizing spanning tree algorithms". *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, 761:400410, Berlin, Germany, December 1993.
- [20] Ted Herman. "Models of Self-Stabilization and Sensor Networks". *International Workshop on Distributed Computing*, 205214, Kolkata, India, 2003.
- [21] George Varghese. "Self-Stabilization by Counter Flushing". *SIAM J. Comput.* 30(2), 486-510 (2000).
- [22] Sukumar Ghosh, Arobinda Gupta, Sriram V. Pemmaraju. "Fault-containing network protocols". *Proceedings of the 1997 ACM symposium on Applied computing*, p.431-437, April 1997, San Jose, California, United States.
- [23] Sukumar Ghosh, Arobinda Gupta, T. Herman, Sriram V. Pemmaraju. "Fault-containing Self-Stabilizing Algorithms". *15th Annual ACM Symposium on Principles of Distributed Computing*, 1996, pp. 45-54.