Survey on Self-Stabilization Concepts and Fault Tolerance

Alexander Dung[1] Thulasiram Duraisamy[2] Sai Raj Reddy Dyappa[3] Samyugdha Radhakrishnan[4]
{alexander.dung[1], thulasiram.duraisamy01[2], sairajreddy.dyapa01[3], samyugdha.radhakrishnan01[4]}
@student.csulb.edu

Pooria Yaghini
Department of Computer Engineering and Computer Science
California State University Long Beach
pooria.yaghini@csulb.edu

***Abstract-* Self-stabilization is a property of certain distributed algorithms that has side effects that are valuable for use in error-prone distributed systems. This paper presents a comprehensive survey of self-stabilization concepts and their role in fault tolerance within distributed systems. Self-stabilization, a property allowing distributed algorithms to recover to a legitimate state after faults, is thoroughly explored. We review the foundational definitions, underlying principles, and the impact of daemons on self-stabilizing algorithms. The survey discusses probabilistic variants, compares self-stabilization with traditional fault-tolerance techniques, and analyzes the performance and challenges of several notable algorithms, such as Dijkstra's Token Ring Algorithm and Uniform Dynamic Leader Election. The paper also explores practical applications, highlighting their potential in dynamic, error-prone environments like ad hoc networks. We conclude by emphasizing self-stabilization's benefits, its current limitations, and promising research directions.**

***Keywords*: Self-stabilization, Fault tolerance, Distributed algorithms, Daemons, Probabilistic convergence, Ad hoc networks, Dijkstra's Token Ring**

1. INTRODUCTION

Algorithms in distributed systems face unique challenges in maintaining stability and reliability compared to programs in other environments. Due to their distributed nature, these algorithms must divide control over each individual node in the system. This has important implications on how these algorithms manage information between parts of the system, and how steps of the algorithm are orchestrated. The distribution of control and information means that preserving the progress of the system when an error occurs in a node becomes a challenge. Not only must the faulty node recover, but we may have to ensure that, for example, the algorithm has not progressed with the assumption that the node was functional, or that the fault has not caused the algorithm to progress incorrectly. If these problems have occurred, then recovery may not be possible, and we risk having to reinitialize the system and start from the beginning again.

In the case of fault tolerance for distributed systems, self-stabilization was proposed as a useful solution. It promises the ability for algorithms to be affected by faults and still eventually reach a valid solution. Originally, it was identified by Dijkstra in 1974 [4] but was disregarded for years. More than a decade later, it was claimed by Leslie Lamport as "a milestone in work on fault tolerance" [12]. Since then, the field has enjoyed a wealth of research into how to construct these algorithms, and their potential applications.

In this survey, we aim to explore the growth in research that the field of self-stabilization has experienced. First, we will explain the key concepts, including definitions of the concept itself, the daemons that play a fundamental part in the function of self-stabilizing algorithms, and a weaker variation of the original definition. Second, we will explain self-stabilization's value as a form of fault tolerance and its role in the suite of fault tolerance techniques. Then, we will explore some examples of self-stabilizing algorithms. Finally, we delve into the applications of these algorithms in the real world.

2. KEY CONCEPTS

Since its original proposal, the definition of self-stabilization has undergone many further clarifications and some variations that prove to be useful. First, we will explore the original definition.

2.1 Original Definition

The concept of self-stabilization was originally presented by Dijkstra [4] based on the concepts of privileges and legitimate or illegitimate states. Privileges are boolean rules that depend on the state of the node and its neighbors, which are used for determining whether that privilege's associated state transition can be taken. The legitimacy of a state of the distributed system is defined by the algorithm designer. For example, for Dijkstra's original token ring system, a state is legitimate if there exists only one privilege in the system.

The rules for the legitimacy of a state are summarized by Dijkstra like so [4]:

> We require that: (1) in each legitimate state one or more privileges will be present; (2) in each legitimate state each possible move will bring the system again in a legitimate state; (3) each privilege must be present in at least one legitimate state; and (4) for any pair of legitimate states there exists a sequence of moves transferring the system from the one into the other.

A system is self-stabilizing if for any starting state, a privilege always exists in the system (there is always another step to take) and the system can reach a legitimate state in a finite number of steps. Dijkstra notes, however, that this definition is based on a model of a distributed system whose nondeterministic execution order uses a central "daemon" to choose a single privileged node to act at each step. The use of this daemon means the model assumes mutual exclusion in the privileged nodes' actions, which cannot always be enforced in distributed systems due to the lack of centralized control.

2.2 Generalized Definition

Schneider proposes another definition of self-stabilization based on two rules given the state of the system S and a predicate P determining the legitimacy of a state [16]:

1. **Closure**—P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.
2. **Convergence**—Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

Schneider's definition is more concise than Dijkstra's while also being applicable to more forms of algorithms, such as those which converge to a solution and stop executing rather than continuing infinitely. This is necessary because many distributed algorithms such as those for graph coloring and independent sets have self-stabilizing algorithms yet reach a static solution and subsequently terminate. Furthermore, this definition more accurately describes self-stabilizing algorithms because it does not assume mutual exclusion between nodes during state transitions. In distributed systems, one cannot expect mutual exclusion between nodes because they would only have knowledge about their own state and their neighbors' states—in situations where multiple non-neighboring nodes are privileged, they cannot be expected to uphold mutual exclusion and wait for the other to transition first.

2.3 Daemons

The daemons mentioned in previous sections play an important part in determining the real-world effectiveness of a self-stabilizing algorithm. Self-stabilizing algorithms are verified on a certain daemon, which makes assumptions about how nodes are scheduled and selected for execution. Different algorithms are developed assuming different daemons, which means that these algorithms are not guaranteed to stabilize if the daemon was changed for another, due to changing the rules on how nodes are scheduled. Dubois and Tixeuil propose a taxonomy for daemons [6], which allows for classification of daemons, and comparison of daemons and algorithms' ability to self-stabilize against different classes of daemons. This taxonomy is based on four numerical attributes: distribution, fairness, boundedness, and enabledness.

2.3.1 Distribution

Distribution characterizes the spatial constraints on scheduling. A daemon is k-central if only nodes that are at least k hops away from each other can be scheduled. Following this definition, a distributed daemon is 0-central because any node is able to be scheduled at any time regardless of other nodes in the graph. A central daemon as used by Dijkstra has centrality equal to the diameter of the graph because no two nodes can be scheduled at the same time. For all $k \geq 0$, the set of k-central daemons contains the set of all $(k+1)$-central daemons. The set of 0-central daemons is equivalent to the set of all daemons. This set relationship is visualized in Figure 1.

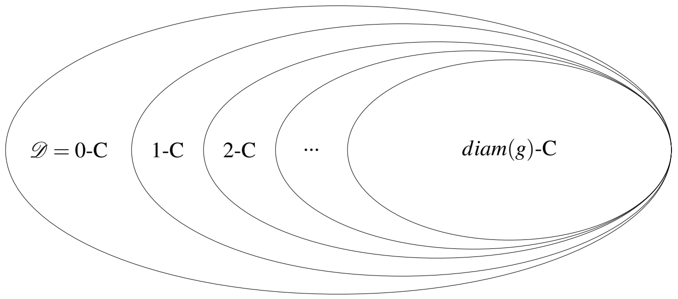

Figure 1: Set relationships of daemons of different distributions [6]

2.3.2 Fairness

Fairness describes the daemon's selection process for activating a node. A daemon can be unfair, weakly fair, strongly fair, or Gouda fair. An unfair daemon does not guarantee that a process that is enabled will ever be scheduled for execution. Weak fairness means that once a process is enabled, it will eventually be scheduled to execute. Strong fairness means that processes that are infinitely enabled will eventually be scheduled to execute. Gouda fairness means that for any configuration, all possible state transitions from that configuration will eventually occur. The daemon that Dijkstra described in his original definition of self-stabilization is an unfair daemon, since it selects among privileged nodes purely randomly. The set of unfair daemons contains the set of weakly fair daemons, which contains the set of strongly fair daemons, which contains the set of Gouda fair daemons. The set of unfair daemons is equivalent to the set of all daemons. This set relationship is visualized in Figure 2.

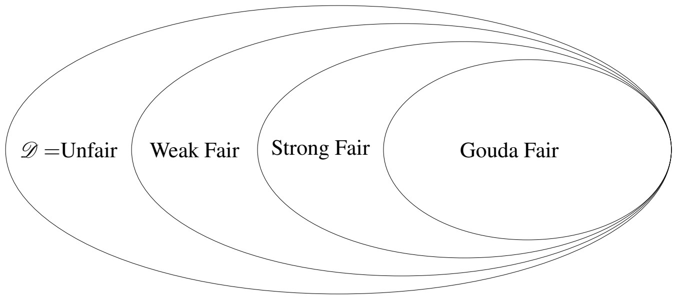

Figure 2: Set relationships of daemons of different fairness [6]

2.3.3 Boundedness

Boundedness describes heterogeneity of scheduled processes. A k-bounded daemon guarantees that any one process cannot be scheduled to execute more than k times between two schedules of any other one process. This does not necessarily mean that a process cannot execute more than k times in a row - a daemon is still k-bounded if, given two processes, one process is never scheduled to execute while the other is continuously scheduled. For all $k > 1$, the set of k-bounded daemons contains the set of $(k-1)$-bounded daemons. The set of ∞-bounded daemons is equivalent to the set of all daemons. This set relationship is visualized in Figure 3.

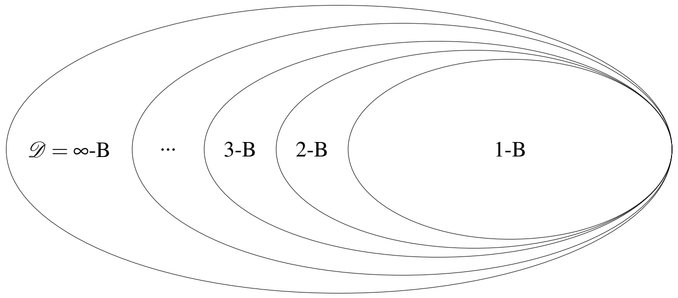

Figure 3: Set relationships of daemons of different boundedness [6]

2.3.4 *Enabledness*

Enabledness describes a bound on how long an enabled process can go without being scheduled. A k-enabled daemon guarantees that a process will be scheduled to execute before it is enabled k times. Note that a nonzero k-enabled daemon implies that the daemon is weakly fair. However, the converse is not true - a weakly fair daemon is not guaranteed to have bounded enabledness. For all $k > 0$, the set of k-enabled daemons contains the set of $(k-1)$-enabled daemons. The set of ∞-enabled daemons is equivalent to the set of all daemons.This set relationship is visualized in Figure 4.

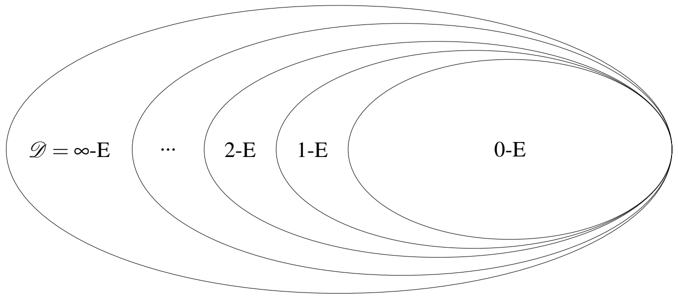

Figure 4: Set relationships of daemons of different enabledness [6]

Dubois and Tixeuil also propose a method of comparing daemons. A daemon's class can be denoted by their four attributes as a quadruple: $d(C,B,E,F)$. If we consider a daemon as the set of all executions it allows, then we hold that daemon d‘ is more powerful than d if d‘ allows all possible executions allowed by d. This implies that the set of all daemons is the strongest class. Also, a daemon $d(C,B,E,F)$ is stronger than d‘$(C$‘$,B$‘$,E$‘$,F$‘$)$ if any of C‘, B‘, E‘, or F‘ are a subset of C, B, E, or F respectively.

We thus can define the most powerful daemon for each class as the canonical daemon for that class, as it allows all possible executions for all daemons of its class. Therefore, if an algorithm can self-stabilize against the canonical daemon of a class, it can self-stabilize against any other daemon of that class, and against any daemon of a weaker class, because the canonical daemon contains the possible executions of all other daemons in its own class and in weaker classes. For example, an algorithm that stabilizes against the canonical unfair daemon can likely stabilize against weakly fair, strongly fair, and Gouda fair daemons, assuming the three other attributes are the same or weaker.

2.4 Probabilistic Self-Stabilization

Probabilistic self-stabilization is a weakened variant of self-stabilization, proposed originally by Israeli and Jalfon [10]. Their proposal holds that for a protocol whose state transitions are randomized functions of the node and its neighbors' states, it is self-stabilizing if:

1. **Closure**—all legitimate states can only transition to other legitimate states; and
2. **Probabilistic Convergence**—for every initial state, the probability of transitioning to a legitimate state eventually reaches 1.

The introduction of probabilistic self-stabilization is useful because it allows the integration of randomness into self-stabilizing algorithms. As noted by Israeli and Jalfon, randomness has many uses for these algorithms.

Randomness can be used to break symmetry, a problem identified initially by Dijkstra. In his original token ring algorithms, symmetry is broken by making the processors nonuniform. In each of the three algorithms, at least one node is given a different behavior from the others. While this is a viable tactic, another problem arises of ensuring that nodes are assigned the correct role in the algorithm. For example, ensuring that only one node in the token ring is considered the bottom node is a problem with a wealth of research on its own—the leader election problem. By introducing randomness to the protocol for state transitions, symmetry is broken without need for organizing nonuniform classes. In fact, it has been found that many problems cannot be solved with deterministic self-stabilization, including unidirectional token rings and vertex coloring [9]. These problems now have many algorithms using probabilistic self-stabilization instead.

3. RELATION TO FAULT TOLERANCE

Traditional fault tolerance approaches such as [3, 15] aim to handle faults by preventing them from causing errors if possible, and if not, hiding their effects from the user. Notably, this causes fault tolerance mechanisms to treat causes of faults individually, as noted by Gouda and Multari in the case of coordination loss [8]. This means that individual protocols must be written for each cause that is identified, each of which could have differing side effects on the program's state afterward, and could lead to not yet-discovered faults to propagate in production environments without code to handle them.

Using self-stabilization as a form of fault tolerance provides a different approach than that of traditional fault tolerance techniques. The properties of self-stabilization guarantee that the system is guaranteed to, from any arbitrary state, eventually converge to a legitimate state. Therefore, if a self-stabilizing system were to be affected by some fault that changed its state, the system is still guaranteed to converge. Unlike traditional fault tolerance approaches, self-stabilization allows faults to occur to the system, and guarantees recovery and convergence afterward. Convergence allows the system to abstract the effects of any faults that occur and use the same recovery steps regardless of the underlying fault type, as long as it only affects the state of the system.

However, self-stabilization alone does not prove to be a silver bullet for fault tolerance. Notably, none of the definitions of self-stabilization that have been presented provides a bound on the number of steps needed for convergence. While convergence is guaranteed, the lack of a bound on the steps taken means that self-stabilization may not be viable for applications that desire a system with the utmost availability. Also, self-stabilization can only handle faults that affect the state of the system, not the behavior—if, for example, the instruction memory of a node is corrupted, then the change in its behavior could affect the behavior of neighboring nodes and eventually obstruct convergence. Finally, self-stabilization provides no guarantee of convergence if the topology of the system is affected. This is corroborated by Lohs et al.'s experiments with topology fluctuations against self-stabilizing algorithms [14], which show that said algorithms fail to converge when affected by constant changes to the topology.

4. ALGORITHMS

Here we provide some examples to demonstrate the behavior and structure of other typical self-stabilizing algorithms.

4.1 Dijkstra's Token Ring Algorithm

The system is correct if there is exactly one token in the ring. Let's have a look at a simple solution. Given an oriented ring, we simply call the clockwise neighbor parent (p), and the counterclockwise neighbor child (c). Also, there is a leader node v_0. Every node v is in a state S(v) ∈ {0, 1, . . . , n}, perpetually informing its child about its state. The token is implicitly passed on by nodes switching state. Upon noticing a change of the parent state S(p), node v executes the following code:

4.1.1 Algorithm

```
1: if $v = v_0$ then
2:     if $S(v) = S(p)$ then
3:         $S(v) := (S(v) + 1) \mod n$
4:     end if
5: else
6:     $S(v) := S(p)$
7: end if
```

Every node apart from leader v_0 will always attain the state of it's parent. Over time, nodes learn the current state of the leader node v_0 as the token circulates around the ring. This happens sequentially, with one node after the other learning the leader's state.

The leader node v_0 may increase its state multiple times without reaching stability. However, at some point, the leader will reach a state 's' that no other node had at time t_0. This is guaranteed to happen eventually since there are n nodes and n states.

Once the leader reaches state s, it cannot push for s + 1 (mod n) until every node (including its parent) has state s. Therefore, the system must stabilize at this point, ensuring that all nodes have synchronized their states.

After stabilization, there will always be only one node changing its state at any given time. This ensures that the system remains in a legitimate and stable state, with no conflicting state changes occurring simultaneously.

4.2 Token Circulation Protocol In Uniform Networks

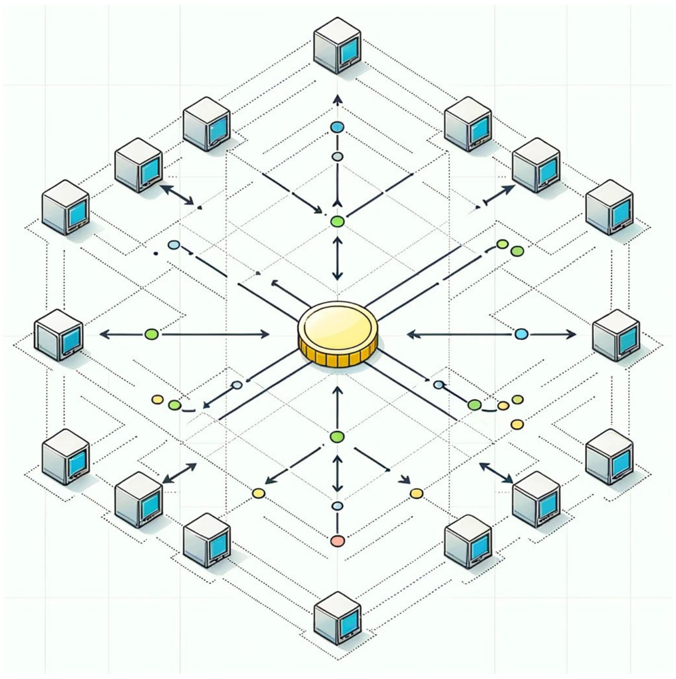

Figure 5: Token Circulation Protocol

The self-stabilizing token circulation protocol [17] addresses the problem of managing token circulation in a network with an arbitrary initial configuration, which may include zero or multiple tokens. The protocol operates in a uniform network of nodes, meaning each node runs the same code and is logically equivalent. This approach is particularly robust against transient faults, which may temporarily corrupt data but do not damage the system permanently. The system's self-stabilization feature ensures that it will eventually reach a stable state where a single token is circulated fairly among all nodes, without needing any manual initialization or external intervention.

Key to the protocol is the mechanism of token passing and tree maintenance within the network. Nodes utilize a "father pointer" to create a dynamic spanning tree structure where the root node (which has a null father pointer) holds the token.

Token circulation is controlled through a three-color scheme, ensuring fair distribution by allowing the token to be passed only to a node whose color is a specific relation to the root's current color. This scheme helps in preventing the token from being monopolized by any subset of nodes and supports continuous operation despite possible transient states induced by faults.

4.2.1 Algorithm

```
1: Variables for each node $i$:
2:     $F_i$: Father pointer, can be NULL for the root.
3:     $C_i$: Color variable, values in $\{0,1,2\}$.
4: Initialization:
5:     Each node $i$ initializes $F_i$ and $C_i$ arbitrarily.
6: procedure PassToken($i$)
7:     if $F_i ==$ NULL then ▷ $i$ is the root and has the token
8:         Find a child $k$ where $C_k = (C_i + 1) \mod 3$
9:         if such $k$ exists then
10:            $F_i = k$                    ▷ Pass the token to child $k$
11:        else
12:            $C_i = (C_i + 1) \mod 3$     ▷ Change color and retry
13:        end if
14:    end if
15: end procedure
16: procedure HandleFaults($i$)
17:    if $F_{F_i} == i$ then
18:        $F_i =$ NULL                     ▷ Break potential cycle
19:    end if
20: end procedure
21: Rules to Execute in Each Node $i$:
22: R1:
23: if $F_i ==$ NULL then
24:     PassToken($i$)    ▷ Only the root with NULL pointer passes the token
25: end if
26: R2:
27: if $F_{F_i} == i$ and $(C_{F_i} == (C_i + 1) \mod 3)$ then
28:     $F_i =$ NULL                        ▷ Become the new root
29: end if
30: R3:
31: if $(F_{F_i} == i)$ and not $(C_{F_i} == (C_i + 1) \mod 3)$ then
32:     $C_i = (C_i + 1) \mod 3$    ▷ Correct color to maintain cycle
33: end if
34: Scheduler:
35: Execute rules R1, R2, and R3 in a fair and serial manner across all nodes.
```

The protocol also includes robust error handling to manage and recover from faults effectively. This involves procedures for merging multiple tree structures into a single spanning tree and ensuring the token's fair circulation continues uninterrupted. The system is designed to operate correctly under a fair scheduler assumption, where each node gets scheduled to perform operations independently. This design allows the network to stabilize from any arbitrary state to a predictable, efficient configuration, showcasing the protocol's resilience and adaptability in maintaining consistent network operations amidst possible disruptions.

4.3 Uniform Dynamic Leader Election

In the leader election problem [5], the primary goal is to elect only one node as the leader in a distributed system and ensure that all nodes recognize this leader. The challenge in a uniform system, where all nodes share identical properties and can change dynamically, is to break the symmetry and ensure that one node is elected as the leader.

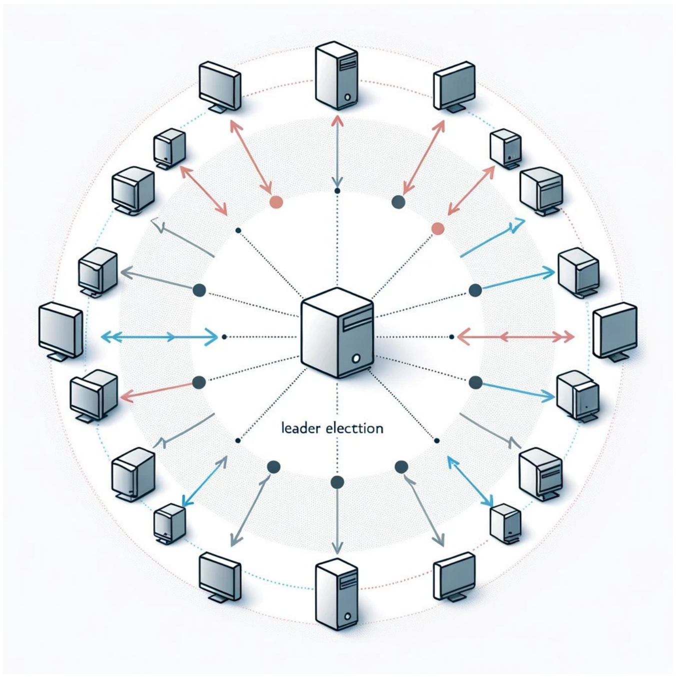

Figure 6: Leader Election Protocol

The algorithm operates in two phases: the cycle elimination phase and the tree fusion phase. In the cycle elimination phase, cycles are eliminated in the "father-son" relationship graph (FSG), ensuring that the resulting subgraph becomes a forest of trees. Here, the father-son relationship is established as each node either becomes a root (leader of its subtree) or has a unique father. Nodes calculate their distance to the root of their subtree, and if a node notices that its distance to the root is growing abnormally, it cuts its connection to its father and becomes a root itself, preventing cycles.

4.3.1 Algorithm

```
1: do forever
2: Reading phase:
3: for $j := 1$ to $A$ do
4:     $(tid_i[j], dis_i[j], f_i[j], color_i[j], ack_i[j], ot_i[j]) :=$ read$(r_j)$
5: end for
6: $max\_tid :=$ max$(tid_i[j])$
7: $min\_dis :=$ min$(dis_i[j] \mid tid_i[j] = max\_tid)$
8: if $(tid_i[f_i], dis_i[f_i]) = (max\_tid, min\_dis)$ then
9:     $(F, C, A, OT) := (f_i, color_i, ack_i, ot_j)$
10: else
11:     $(F, C, A, OT) := (color_i[F],$ false, false$)$
12: end if
13: if $(tid, dis_i) \geq (max\_tid, min\_dis)$ then
14:     write $(tid, dis_i, f_i, color_i, ack_i, ot_i) := (max\_tid, min\_dis+$
    1, $F, C, A, OT$)
15: end if
16: if $\forall j$ such that $(tid_i[j] = tid_i)$ and $(|dis_i - dis_i[j]| \leq 1)$ then
17:     write $(dis_i, f_i) := (0, \text{NULL})$            ▷ become a root
18: end if
19: Cycles elimination
20: procedure TREE-FUSION
21:     if $dis_i = 0$ and $\forall j$ such that son$(j) \Rightarrow ((color_i[j] = color_i)$ and $ack_i[j])$ then
22:         write $(tid_i, ack_i) := (\text{extend\_tid}(tid_i, \text{false}))$
23:     end if
24:     if $\exists j$ such that $(ot_i[j] = \text{true})$ or $(color_i[j] \neq color_i)$ then
25:         write $color_i := \text{choose\_color}(previous\_color, color_i)$
26:     else
27:         write $ot_i := \text{false}$
28:     end if
29:     if $dis_i \neq 0$ then
30:         if $color_i \neq color_i[f_i]$ then
31:             if $\exists j$ such that $color_i[j] \neq (color_i, color_i[j])$ then
32:                 write $ot_i := \text{true}$
33:             end if
34:         end if
35:         write $(color_i, ack_i) := (color_i[f_i], \text{false})$
36:     else
37:         if $\neg ack_i$ and $\forall j$ such that son$(j)$ and $(color_i[j] = color_i$ and $ack_i[j])$ then
38:             write $ack_i := \text{true}$ $\exists j$ such that son$(j)$ and $ot_i[j] = \text{true}$
39:             write $ot_i := \text{true}$
40:         end if
41:     end if
42: end procedure
```

Randomization is used in identifier extension to prevent collisions and ensure unique identifiers. Non-root nodes copy their father's identifier, and nodes without a father (roots) extend their identifiers.

Global synchronization ensures consistent propagation of changes across the tree structure, allowing nodes to participate in the global synchronization protocol.

The stabilization time required for the algorithm to stabilize and elect a leader depends on the graph's maximal degree (A), diameter (D), and the number of nodes (n). The algorithm stabilizes in $O(AD \log n)$ time if n is unknown, and $O(AD)$ otherwise.

5. APPLICATIONS

The benefits provided by self-stabilization lends it to potential use in network applications such as wireless sensor networks and ad hoc networks. It is especially desirable because the definition of convergence implies that a distributed system can start from any configuration without any special initialization procedures. For network applications, this means that if the system experiences a fault, there is no need to stop the system and reinitialize every node. Instead, the system can continue executing the algorithm and the node that faulted will restart and eventually rejoin the system. Moreover, self-stabilization enhances resilience and adaptability in networks by ensuring that faulty nodes recover autonomously and reintegrate into the system without human intervention. This continuous recovery process minimizes downtime and allows for uninterrupted service, which is crucial for applications requiring high availability. In wireless sensor networks, self-stabilization enables nodes to detect and correct errors locally, significantly reducing the need for centralized management. Furthermore, its ability to handle unexpected failures or configuration changes makes it an excellent choice for dynamic and resource-constrained environments, like ad hoc networks. By incorporating self-stabilization, network applications can effectively maintain robust communication even in challenging conditions, fostering greater network longevity and reliability.

5.1 Ad Hoc Networks

Ad hoc networks are networks of devices with no communication infrastructure between them, such as routers or switches. Each node in the network directly communicates with others in its range, meaning communication with nodes outside of range must hop from node to node. Self-stabilization has been a topic of discussion in research for ad hoc networks and wireless sensor networks because of these networks' tendency to experience transient faults due to factors such as radio transmission errors, hardware failure, and harsh environments in the case of wireless sensors. Self-stabilization promises a strong tolerance to these errors and guarantees recovery without needing to stop and reinitialize other nodes in the network. Because of this, self-stabilizing algorithms for ad hoc network applications like link monitoring [18], unicast routing [7, 11], and network structure/clustering [1, 2] are an ongoing area of research. On the other hand, criticisms in this field state that proposed self-stabilizing algorithms are often proven with little regard for real-life factors like radio models, message loss, and corruption [14], and that the lack of a bound for convergence makes recovery time unpredictable [13].

6. CONCLUSION

We explored self-stabilization, a concept for distributed algorithms that provides powerful benefits for algorithms built around it. It allows distributed systems to guarantee recovery from transient faults, and allows the algorithm to abstract these transient faults such that the same recovery procedures can be applied regardless of the effects of the fault. Research into these self-stabilizing algorithms has shown that these algorithms' ability to stabilize is heavily dependent on the environment in which they execute, as shown by the effects of daemons on different algorithms. We also showed that self-stabilization's strengths lend itself as a powerful, but still limited, approach to fault tolerance, which can be exploited in environments that are prone to faults such as ad hoc networks.

7. ACKNOWLEDGMENTS

This document is derived from previous conferences, in particular MICRO 2013, ASPLOS 2015, MICRO 2015, MICRO 2016, MICRO 2017, MICRO 2018, MICRO 2019, MICRO 2020 and MICRO 2021, as well as SIGARCH/TCCA's

Table 1: Comparison of Self-Stabilizing Algorithms

Feature	Dijkstra's Token Ring	Token Circulation Protocol	Uniform Dynamic Leader Election
Network Structure	Oriented Ring	Arbitrary, Uniform	Arbitrary Graphs
Circulation/Election Method	Explicit State Incrementing	Color-Based Token Passing	Identifier-Based Leader Coordination
Fault Tolerance	Resets to Correct Token State	Handles Zero or Multiple Tokens	Handles Node and Link Failures
Stabilization Time Complexity	$\mathscr{O}(n)$	$\mathscr{O}(d)$ (Tree Diameter)	$\mathscr{O}(AD\log n)$ or $\mathscr{O}(AD)$
Memory Usage per Node	Constant Memory	Small Memory for Parent Pointers	Variable, Depends on Identifier Length
Convergence Approach	Cyclic Propagation	Hierarchical Synchronization	Tree Fusion after Cycle Elimination
Distinct Features	Leader Incrementally Adjusts Token	Self-Stabilizing Spanning Tree	Randomization to Break Symmetry
Global Synchronization	N/A	Father-Child Hierarchy	Protocol Coordinates State Across Tree
Legitimacy Conditions	One Token Circulates	One Token Circulates Fairly	Single Leader Controls Entire Network

Recommended Best Practices for the Conference Reviewing Process.

REFERENCES

[1] M. Ba, O. Flauzac, B. S. Haggar, F. Nolot, and I. Niang, "Self-stabilizing k-hops clustering algorithm for wireless ad hoc networks," in *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ser. ICUIMC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2448556.2448594

[2] D. Bein, A. Datta, C. Jagganagari, and V. Villain, "A self-stabilizing link-cluster algorithm in mobile ad hoc networks," in *8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN'05)*, 2005, pp. 6 pp.–.

[3] F. Cristian, "A rigorous approach to fault-tolerant programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 23–31, 1985.

[4] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, p. 643–644, nov 1974. [Online]. Available: https://doi.org/10.1145/361179.361202

[5] S. Dolev, A. Israeli, and S. Moran, "Uniform dynamic self-stabilizing leader election," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 4, pp. 424–439, 1997. [Online]. Available: https://doi.org/10.1007/BFb0022445

[6] S. Dubois and S. Tixeuil, "A taxonomy of daemons in self-stabilization," 2011.

[7] H. T. Evcimen, V. K. Akram, and O. Dagdeviren, "Performance evaluation of distributed self-stabilizing dominating set algorithms in wireless sensor networks," in *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, 2018, pp. 428–432.

[8] M. Gouda and N. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 448–458, 1991.

[9] M. Gradinariu and S. Tixeuil, "Self-stabilizing Vertex Coloring of Arbitrary Graphs," in *International conference on Principles of Distributed Systems (OPODIS 2000)*, Paris, France, Dec. 2000, pp. 55–70. [Online]. Available: https://hal.sorbonne-universite.fr/hal-00631707

[10] A. Israeli and M. Jalfon, "Token management schemes and random walks yield self-stabilizing mutual exclusion," in *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 119–131. [Online]. Available: https://doi.org/10.1145/93385.93409

[11] R. Khot, R. Poola, K. Kothapalli, and K. Srinathan, "Self-stabilizing routing algorithms for wireless ad-hoc networks," in *Distributed Computing and Internet Technology: 4th International Conference, ICDCIT 2007, Bangalore, India, December 17-20. Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 54–66. [Online]. Available: https://doi.org/10.1007/978-3-540-77115-9_5

[12] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency," *SIGOPS Oper. Syst. Rev.*, vol. 19, no. 4, p. 34–44, oct 1985. [Online]. Available: https://doi.org/10.1145/858336.858339

[13] S. Lohs, J. Nolte, G. Siegemund, and V. Turau, "Self-stabilization - a mechanism to make networked embedded systems more reliable?" in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016, pp. 317–326.

[14] S. Lohs, G. Siegemund, J. Nolte, and V. Turau, "Influence of topology-fluctuations on self-stabilizing algorithms," in *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2016, pp. 122–124.

[15] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, p. 222–238, aug 1983. [Online]. Available: https://doi.org/10.1145/357369.357371

[16] M. Schneider, "Self-stabilization," *ACM Comput. Surv.*, vol. 25, no. 1, p. 45–67, mar 1993. [Online]. Available: https://doi.org/10.1145/151254.151256

[17] L.-C. W. Shing-Tsaan Huang, "Self-stabilizing token circulation in uniform networks," *Springer*, vol. Distrib Comput 10, no. 181–187, July 1997. [Online]. Available: https://doi.org/10.1007/s004460050035

[18] Y. Yigit, C. U. Ileri, and O. Dagdeviren, "Fault tolerance performance of self-stabilizing independent set algorithms on a covering-based problem: The case of link monitoring in wsns," in *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, 2018, pp. 423–427.