Simulation Experiments of a Distributed Fault Containment Algorithm Using Randomized Scheduler

Anurag Dasgupta
Computer Science
Valdosta State University
Valdosta, USA
adasgupta@valdosta.edu

David Tan
Computer Science
Valdosta State University
Valdosta, USA
dqtruong@valdosta.edu

Koushik Majumder
Computer Science & Engineering
Maulana Abul Kalam Azad
University of Technology
India
koushik@ieee.org

***Abstract*—Fault containment is a critical component of stabilizing distributed systems. A distributed system is termed stabilizing (or self-stabilizing) if it exhibits two properties – a) convergence: a finite sequence of moves leading to a stable configuration, and b) closure: the system remains in that legitimate state unless another fault hits. In today's world, the likelihood of several failures is quite low, and a single fault is far more likely to occur. The results of simulation experiment we did for single failure instances are presented in this study. For node selection, we employed a randomized scheduler. The results show that the fault containment mechanism we used restores valid configurations after a transient fault. The studies took into account variations in the number of nodes and the degree of the malfunctioning node. The results were graphically and numerically presented to provide relevant information. We can learn about the efficiency of our method by analyzing the simulation outcomes.**

Keywords— Stabilization, simulation, Distributed Systems, Fault Containment, Algorithm

I. Introduction

Fault containment is a critical feature of stabilizing distributed systems. Random illegitimate configurations can occur because of transient failures that can disrupt the system state. A distributed system is said to be stabilizing, when from any given configuration, there exists a sequence of moves, using which the system can reach a legitimate configuration, and once it reaches that configuration, it stays in that state unless any other fault occurs in the system [1, 2].

In most modern day systems, the probability of a massive failure is considered low and single fault is much more likely to occur compared to multiple failures. Containing single failures is more important these days because of improved system reliability. Local transient faults are likely to give rise to almost legal states, instead of arbitrary states. To make sure that non-faulty processes mostly stay unaffected by such local failures, some algorithms allow just a small section of the network around the faulty node to make state changes. The authors in [3, 4] describe how fault containment followed by recovery is achieved using this strategy.

In our fault containment algorithm, weak stabilization is considered. Weak stabilization is defined in the following way - starting from an arbitrary configuration, there exists at least one computation that leads the system to a legal configuration. Weak stabilization algorithms are best implemented using a randomized scheduler [5, 6]. Therefore, our algorithm is going to be implemented using a randomized scheduler to guarantee eventual recovery.

In this paper, the authors wrote programs to implement a fault containment algorithm and run simulation experiments. The simulations provided insight regarding the efficiency of the algorithm. The authors experimented by varying the number of nodes and by changing the faulty node's degree.

A fundamental understanding of graph theory is assumed and relevant notations are used throughout the paper to discuss the algorithm of self-stabilization and its results. The readers are recommended to consult [7] for an overview of graph theory and the notations and definitions used in this paper.

II. Background

A. Motivation

The occurrence of a transient fault could be catastrophic in certain systems and conditions and it can be very expensive to fix the fault. Sometimes it could be almost impossible too to manually rectify the fault. Consider the example of a commercial satellite orbiting around the Earth. If the shuttle's power supply were to fail due to some electrical malfunction, a major investment of time, effort, and resources would be wasted. A few such examples are – submarine communications cable system under the ocean [8], systems in satellite or space stations [9], sensors in battle fields [10] etc.

One way to resolve the above problem is to design the system in such a way so that the satellite or the cable network would self-stabilize in response to the fault. In other words, this means after a certain period of time the system would get back to what is known as legal states and resume its normal operations without any outside intervention. This is the general idea of autonomous fault containment and because of the nature of the solution, its practical implications are of great significance.

We can generalize this concept to hold for any random configuration of any such system. To continue with our examples, consider the satellite or the cable network at any

The authors would like to thank Blazer Summer Research Institute (BSRI) of Valdosta State University for supporting this research in the summer of 2021.

stage in their state-space tree, and after self-stabilization, we end up with a *legal configuration* of the system. This is the main advantage of self-stabilizing algorithms and it is for this characteristic that self-stabilization is a much-sought property in modern distributed systems.

B. Related Work

Dijkstra's landmark paper on stabilization in 1974 was the first paper in the field of self-stabilization [2]. Since then, stabilization research has come a long way. Lately stabilization research not only is limited to computer science alone, but also combines other fields such as mathematics, biology, engineering, physics etc. For the interested readers, some important applications like the construction of stabilizing spanning trees in a graph [11], stabilizing maximum flow trees in network topology [12], construction of stabilizing rooted shortest path tree [13], stabilizing maximal matching algorithm [14], stable path problem [15] etc. are suggested. For readers who need an introduction and basic understanding of the subject, an excellent and lucid introduction might be the book written by Dolev [1].

The probabilistic fault containment algorithm based on which we conducted our experiments in this paper was derived by Dasgupta et al. [16]. In this paper, a thorough and elaborate explanation of the stabilization algorithm is provided along with formal proofs of correctness and other important results.

The current paper features programs written in Java of the proposed algorithm and tested on varying instances of number of nodes and the degree of the faulty node. We record the time taken, and cite interesting observations made from the results along with implementation details.

C. Model and Notation

For the sake of simplicity, we only considered simple connected graphs, i.e., graphs without parallel edges and self-loops.

Definitions:

Given a graph $G = (V, E)$, the *neighborhood* N_i of a vertex, $v_i \in V$, is defined to be $\{v_j \in V \mid \exists(v_i, v_j) \in E\}$, $i \neq j$, i.e., the neighborhood of a vertex constitutes all vertices that share an edge with the vertex.

Each vertex $v_i \in V$ holds two variable values: a primary variable value, $v(i)$, and a secondary variable value, $x(i)$. The primary value of the vertices can be either 0 or 1, i.e., $v(i) \in \{0, 1\}$ whereas the secondary value $x(i) \in \mathbb{Z}^+$. This approach helps model our system to elect local leaders as described in [16].

The *local leader* is defined as the vertex with the largest secondary variable value amongst all vertices in its neighborhood. Formally, the local leader is a vertex $v_k \in N_i \cup v_i$ such that $\forall v_j \in N(i) \cup v_i$, $x(j) \leq x(k)$.

For our fault containment algorithm, a constant arbitrarily large positive integer, $m \in \mathbb{Z}^+$ is required. This is used to increase the weight of the local leader. This variable acts like a fence and hence helps in containing the fault. Only when some other node in its neighborhood is chosen more than a certain value determined by m, a possible new leader is elected.

III. Algorithm

The algorithm assumes a randomized scheduler which picks a node or a process represented by a vertex i and compares the state of i with the state of its neighbors, $N(i)$. This is known as the shared memory model. If the selected vertex satisfies one of the algorithm rules, the state of the vertex is *updated*. If the chosen vertex does not match a rule, the state of the vertex remains the same.

$$\forall j \in N(i),\ v(i) \neq v(j) \Rightarrow \text{Update } v(i) \qquad (1)$$

Rule (1) states that if all neighboring vertices have a different primary variable value than the selected vertex, the primary variable value of the vertex will be updated to match that of its neighbors.

$$\exists j \in N(i) \text{ s.t. } v(i) \neq v(j) \wedge \neg(\forall j \in N(i),\ v(i) \neq v(j)) \wedge i \text{ is the local leader} \Rightarrow \text{Update } v(i) \wedge x(i) = x(i) + m \qquad (2)$$

Rule (2) states that if there is a neighbor with a different primary variable value than the chosen vertex, and Rule 1 is not satisfied, and the chosen vertex is the local leader - then the chosen vertex's primary variable value is updated and the secondary variable value of the vertex is increased by m.

$$\exists j \in N(i) \text{ s.t. } v(i) \neq v(j) \wedge \neg(\forall j \in N(i),\ v(i) \neq v(j)) \wedge i \text{ is not the local leader} \Rightarrow x(i) = x(i) + 1 \qquad (3)$$

Rule (3) states that if there is a neighbor with a different primary variable value than the chosen vertex, and Rule 1 is not satisfied, and the vertex is not the local leader - then the secondary variable value of the vertex is increased by 1 but the primary variable value of the vertex remains unchanged.

Rule (2) increases the secondary variable value of the local leader by m while flipping the leader's primary variable value. But if all the neighbors have a different primary variable value, in that case the algorithm flips the vertex's primary variable value without increasing the secondary variable value. The methodology of increasing a chosen vertex's secondary variable value ensures that the probability of the fault propagation is very low and constitutes the heart of the algorithm.

IV. Methodology

The approaches and strategies utilized to construct the code in the research experiment and simulations will be covered in this part.

Eclipse was our IDE (integrated development environment). Due to the faster buffer time when executing the code, we decided to switch to Intelli J [17] throughout the experiments. The overarching goal was to record the time spent within the fault gap, which is defined as the shortest interval after which the system was ready to handle the next single failure with the same level of efficiency [16]. Other resources we used to build this sophisticated application include Coursera [18] and edx [19].

We created code for a random graph generator in the application since we had to construct any random graph with provided node counts. Fig. 1 illustrates a piece of code that demonstrates our program's "Graph" class

```
public class Graph {

    public static void main(String[] args) {
        ArrayList<Integer> x_values = new ArrayList<>();
        int size = 21;
        for (int i = 0; i < size; i++) {
            x_values.add(i);
        }
        Collections.shuffle(x_values);
        ArrayList<Node> nodes = new ArrayList<>();
        // generate 20 nodes
        for (int i = 0; i < size; i++) {
            nodes.add(new Node(x_values.get(i)));
        }
        // link them randomly
        linking(nodes);

        long start = System.currentTimeMillis();
        String firstTime = java.time.LocalTime.now().toString();

        Node finalNode = processing(nodes);

        System.out.println("Time start: " + firstTime);
        System.out.println("Time end: " + java.time.LocalTime.now());
        System.out.println("Total time taken for execution: "
                + (System.currentTimeMillis() - start) + " milli second");

        System.out.println("Node " + finalNode.getNumber() + " was selected.");
        System.out.println(finalNode.connection());

    }
```

Fig. 1. Code snippet showing the main method of the program.

The program outputs how much time it takes, when the code started, and ended. We added an x value for the secondary variable in the algorithm for all the nodes and their neighbors. The x values are randomized from 0 to 20. Next, we created the randomize scheduler and the scheduler picked any node at random. Then it checks whether the three conditions (rules) of the algorithm for that node are satisfied or not.

This set the stage for the fault containment algorithm. In the end, the stabilization time was recorded. In Fig. 2, the true and false values act as the primary variable. By achieving stabilization, we were able to record the time within the fault gap.

```
public static boolean isAllTrue(ArrayList<Node> nodes) {
    for (Node node : nodes) {
        if (node.getState() == false) {
            return false;
        }
    }
    return true;
}
```

Fig. 2. The Boolean method that contains the primary variable of the nodes.

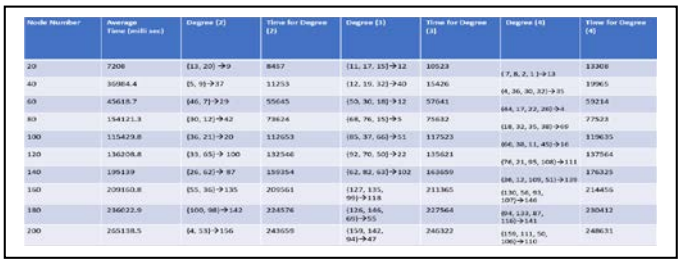
There are three ways for the code to achieve stabilization: first whatever node is selected to be the faulty node, if it is the local leader in the list then the Boolean variable will automatically turn true and this will rectify the fault right away but if it is not the local leader then it will move to step 2 or 3. Step 2 suggests that if one of the neighboring node's primary variable is not equal to that of the faulty node, then its x value will increase by *m* and the primary variable will flip (if it is true then it flips to false and vice versa) until the faulty node becomes the local leader. Step 3 suggests that if the neighboring node's x value is less than or equal to that of the faulty node then the neighboring node's x value will increase by 1 and its primary variable value stays unchanged. This will go on until the faulty node becomes the local leader and thus eventually its primary variable value flips to the same value as its neighbors.

V. RESULTS

This section presents the results and the inferences we made from the results.

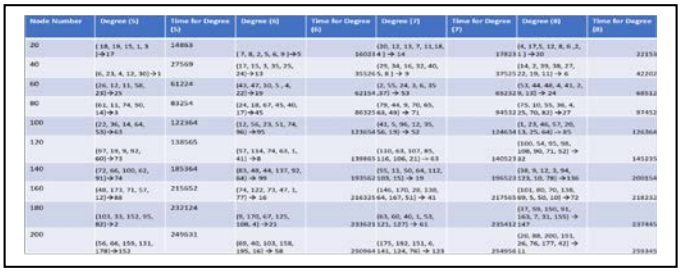
Node Number	20	40	60	80	100	120	140	160	180	200
	26579	55572	72356	110259	132021	153226	210065	221569	243962	277231
	4207	41235	55465	63569	124436	110363	196362	206654	223651	256213
	3477	22578	52365	72569	105569	123554	153265	215565	241623	263816
	8791	32152	12258	82246	113659	151235	206543	213326	235663	271621
	9274	38426	36542	100698	98625	136854	192236	196354	243316	243165
	80	42315	39569	95562	105378	110369	206321	205698	233656	269563
	853	21589	71125	73256	121369	146559	194563	220364	237128	275363
	819	37856	46589	83465	115639	142396	186336	199856	231236	265436
	17971	51582	44569	103356	126963	152646	201136	200656	229633	256312
	29	26539	25349	756233	110639	134886	204563	211566	240361	273165
Total	72080	369844	456187	1541213	1154298	1362088	1951390	2091608	2360229	2651385
Average	7208	36984.4	45618.7	154121.3	115429.8	136208.8	195139	209160.8	236022.9	265138.5

Fig. 3. Average stabilization time (milliseconds) with increase in node number.



Node Number	Average Time (milli sec)	Degree [2]	Time for Degree [2]	Degree [3]	Time for Degree [3]	Degree [4]	Time for Degree [4]
20	7208	{13, 20} →9	8457	{11, 17, 15}→12	10523	{7, 8, 2, 1}→13	12308
40	36984.4	{5, 9}→37	11253	{12, 19, 32}→40	15426	{4, 36, 30, 32}→35	19965
60	45618.7	{46, 7}→19	55645	{55, 36, 18}→12	57641	{44, 17, 22, 28}→4	59214
80	154121.3	{80, 12}→42	73624	{68, 76, 15}→5	75632	{18, 32, 35, 38}→69	77523
100	115429.8	{36, 21}→20	112653	{85, 37, 66}→51	117523	{66, 38, 11, 45}→16	119635
120	136208.8	{93, 65}→ 100	132546	{92, 70, 50}→22	135621	{74, 21, 95, 108}→111	137564
140	195139	{26, 62}→ 87	159354	{62, 82, 63}→102	163659	{26, 12, 109, 51}→120	176325
160	209160.8	{55, 36}→135	205561	{127, 135, 99}→118	211365	{130, 56, 93, 107}→146	214455
180	236022.9	{100, 98}→142	224576	{126, 166, 69}→55	227564	{84, 122, 87, 116}→141	230412
200	265138.5	{4, 33}→156	243659	{159, 142, 94}→47	246322	{156, 111, 56, 106}→110	248631

Fig. 4. Average stabilization time with deviation in faulty node's degrees 2 to 4 and the increase of nodes.



Node Number	Degree (5)	Time for Degree (5)	Degree (6)	Time for Degree (6)	Degree (7)	Time for Degree (7)	Degree (8)	Time for Degree (8)
20	{18, 19, 15, 1, 3 }→17	14863	{7, 8, 2, 5, 6, 9}→5		{20, 12, 13, 7, 11,18, 16023 4 } → 14		{4, 17,5, 12, 8, 0 ,2, 17823 1 }→10	22153
40	{6, 23, 4, 12, 30}→1	27569	{17, 15, 3, 35, 25, 24}→13		{29, 34, 16, 32, 40, 35126 5, 8 } → 9		{14, 2, 39, 38, 27, 37125 22, 19, 11} → 6	42202
60	{26, 12, 11, 56, 28}→25	61224	{43, 47, 10, 5, 4, 22}→19		{2, 55, 24, 3, 6, 35 62154 ,37} → 53		{53, 44, 48, 4, 41, 2, 63232 9, 13} → 24	68512
80	{61, 11, 74, 30, 14}→3	83254	{24, 18, 67, 45, 40, 17}→45		{79, 44, 9, 70, 65, 8032768, 49} → 71		{75, 10, 55, 36, 4, 94532 25, 70, 82} →27	97452
100	{22, 36, 14, 64, 53}→63	122364	{12, 56, 73, 51, 74, 96} →95		{41, 5, 96, 12, 35, 1236564 56, 19} → 52		{5, 23, 46, 57, 20, 1246543 3, 25, 64} → 85	126364
120	{97, 19, 9, 92, 60}→73	138565	{57, 114, 74, 63, 1, 41} →8		{110, 63, 107, 85, 139865 116, 106, 21} → 63		{100, 54, 95, 98, 108, 90, 71, 52} → 1405232	145235
140	{72, 66, 100, 62, 91}→34	185364	{83, 48, 44, 137, 92, 68} → 98		{55, 13, 50, 64, 112, 193562 103, 15} → 10		{38, 9, 12, 3, 94, 1965123 123, 53, 78} →136	200154
160	{48, 173, 71, 57, 12}→88	215652	{74, 122, 73, 47, 1, 77} → 16		{146, 170, 20, 130, 2163264, 167, 51} → 41		{103, 80, 70, 138, 2175659, 5, 50, 10} →72	218232
180	{103, 33, 152, 95, 82}→2	232124	{9, 170, 67, 125, 108, 4} →21		{63, 60, 46, 1, 53, 233623 121, 127} →61		{37, 59, 150, 91, 163, 7, 31, 155} → 235412 147	237485
200	{56, 66, 159, 131, 178}→152	249631	{69, 40, 103, 158, 195, 16} → 58		{175, 192, 151, 6, 250964 141, 124, 76} →123		{20, 88, 200, 193, 26, 76, 177, 42} → 254956 11	259345

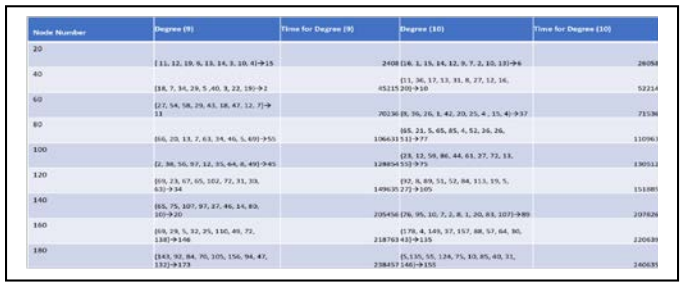
Fig. 5. Average stabilization time with deviation in faulty node's degree 5 to 8 and the increase of nodes.

Fig. 3 displays the different numbers of nodes ranging from 20 to 200 in increments of 20. In our tests, we changed the numbers of nodes to display the scalability of the program. Each time we ran the program, it displayed the stabilization time within the fault gap. Also, we wanted to know the average time it took overall so we ran the code ten times for each of the numbers and thus calculated the average time. What we can infer from the tests is that as predicted – while the scale became larger so did the average time it took for stabilization to occur.

Fig. 4 shows the stabilization time for the faulty node's deviation in degrees 2, 3, and 4 for total number of nodes 20 to 200, by increments of 20. As mentioned previously, the average time was found by producing 10 experimental run times for each of the selected node numbers and averaging ten runs for each node. The value for each degree was randomly generated (in Fig. 4, 5, and 6; indicated within curly braces are the node numbers selected by the program) as described in the methodology portion of the research. The randomized values for each degree was used to run the program and produce 10 experimental run times for each degree. The 10 run times for each degree were averaged as well.

Fig. 5 depicts the data for the deviation in degrees 5, 6, 7, and 8 for number of nodes varied from 20 to 200, by increments

of 20. Figure 6 depicts the data for the deviation in degrees 9 and 10 for number of nodes varied from 20 to 200, by increments of 20. We noticed that the stabilization time increased with the increase in degrees. This was also expected as the program had to check through more neighbors as the degree increased and therefore the stabilization took longer to occur.



Node Number	Degree (9)	Time for Degree (9)	Degree (10)	Time for Degree (10)
20	[11, 12, 19, 9, 13, 14, 3, 10, 4]→15		2408 [18, 1, 15, 14, 12, 9, 7, 2, 10, 13]→6	26058
40	[18, 7, 34, 29, 5 ,40, 3, 22, 19]→2		[11, 36, 17, 13, 31, 8, 27, 12, 16, 45215 20]→10	52214
60	[27, 54, 58, 29, 43, 18, 47, 12, 7]→ 13		70236 [8, 36, 26, 1, 42, 20, 25, 4 , 15, 4]→37	71536
80	[66, 20, 13, 7, 63, 34, 46, 5, 69]→55		[65, 21, 5, 65, 85, 4, 52, 26, 26, 106631 51]→77	110961
100	[2, 38, 56, 97, 12, 35, 64, 8, 49]→45		[23, 12, 59, 86, 44, 61, 27, 72, 13, 128654 55]→75	130513
120	[69, 23, 67, 65, 102, 72, 31, 33, 63]→34		[92, 8, 89, 51, 52, 84, 113, 15, 5, 149635 27]→105	151085
140	[65, 75, 107, 97, 27, 46, 14, 83, 10]→20		205456 [76, 95, 10, 7, 2, 8, 1, 20, 83, 107]→89	207626
160	[69, 29, 5, 32, 25, 110, 49, 72, 138]→106		[178, 4, 149, 37, 157, 88, 57, 64, 30, 218763 43]→135	220639
180	[143, 92, 84, 70, 105, 156, 94, 47, 132]→173		[5,130, 55, 124, 75, 33, 85, 40, 31, 238457 146]→155	240635

Fig. 6. Average stabilization time with deviation in faulty node's degrees 9, 10 and the increase of nodes.

In Fig. 7, the trend of average time with increase of number of nodes is demonstrated. The relationship shown in the figure appeared to be a positive linear relationship between node number and average run time. The relationship indicates that as the node number increases so does the average stabilization time. The graph also revealed some spikes between 60 and 80 and again in the range of 140-150. Besides those deviations, it shows the increase in the average time, when scaling to higher nodes, follows approximately a linear trend. We observe that this positive linear relation ties along with the node numbers as we scale the node numbers higher and higher.

Fig. 8 shows the trend in deviation in degrees of node 20 to 200, by increments of 20. The relationship shown in the figure appeared to be a positive linear relationship between node number and run time for each degree. The relationship indicates that as the node number increases so does the run time for each degree. As we expected, when we increase the degrees of the faulty node, the algorithm takes more time to fully execute because it takes longer time to compare and run through the algorithm rules.

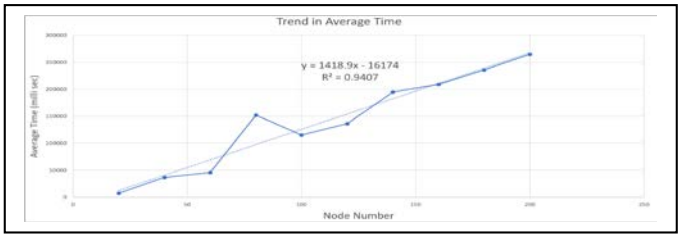


Fig. 7. Trend in average time of selective node numbers.

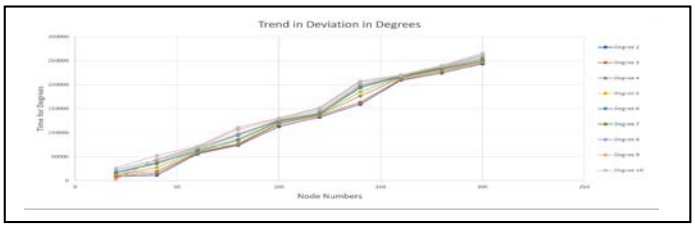


Fig. 8. Trends in deviation in degrees of selective node numbers.

VI. Conlcusion

From our simulations it is evident that our algorithm is effective in resolving faults of transient nature. In this paper, we conducted simulation experiments for single fault scenarios using a randomized scheduler. The simulation results provided insight into the efficiency of our algorithm. We varied the number of nodes and the degree of the faulty node and the obtained results are graphically and numerically analyzed that led to meaningful inference

References

[1] S. Dolev, *Self-stabilization* (MIT Press, 2000).

[2] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, *Comm. ACM 17*, 1974, 643-644.

[3] A. Gupta. Fault-containment in self-stabilizing distributed systems. *PhD thesis*, Department of Computer Science, The University of Iowa, Iowa City, IA, 1997.

[4] A. Dasgupta, Extensions and refinements of stabilization. *PhD thesis*, Department of Computer Science, The University of Iowa, Iowa City, IA, 2009

[5] S. Devismes, S. Tixeuil, M. Yamashita, Weak vs. self vs. probabilistic stabilization, Distributed Computing Systems, International Conference, vol. 0, 2008. pp. 681–688, Los Alamitos, CA.

[6] A. Dasgupta, S. Ghosh, X. Xiao, Fault-containment in weakly-stabilizing systems, *Special Issue: Self-* Systems of Theoretical Computer Science*, 2011.

[7] N. Deo, *Graph theory with applications to engineering and computer science* (Prentice-Hall, 1974).

[8] J. Andres, T. Fang, M. Nedbal, Cable route planning and installation control: recent advances. Sub Optic 2007: Baltimore, Maryland.

[9] M. Fayyaz, T. Vladimirova, Fault-tolerant distributed approach to satellite on-board computer design. 2014 IEEE Aerospace Conference, Big Sky, Montana.

[10] Y. Dalichaouch, P.V. Czipott, A.R. Perry, magnetic sensors for battlefield applications. Aerospace/Defense Sensing, Simulation, and Controls 2001, Orlando, Florida.

[11] N.S. Chen, H.P. Yu, S.T. Huang, A self-stabilizing algorithm for constructing spanning trees, *Information Processing Letters 39*, 1991, 147-151.

[12] A. Dasgupta, Selfish stabilization of maximum flow tree for two colored graphs, *The Pennsylvania Association of Computer and Information Science Educators*, California, PA, 2014.

[13] J. Cohen, A. Dasgupta, S. Ghosh, & S. Tixeuil, An exercise in selfish stabilization. *ACM TAAS, 3*(4): 2008, 1-12.

[14] S. Hsu, S. Huang, A self-stabilizing algorithm for maximal matching. Inf. Process. Lett., 43(2):77–81, 1992.

[15] J. A. Cobb, M. G. Gouda, & R Musunari, A stabilizing solution to the stable path problem, *Self-Stabilizing Systems*, San Francisco, CA, 2003, 169-183.

[16] A. Dasgupta, S. Ghosh, X. Xiao, Probabilistic fault-containment, *Proceedings of the 9th international conference SSS*, Paris, France, 2007, 189-203.

[17] JetBrains IntelliJ IDEA coding assistance and ergonomic design developer productivity website. Retrieved April 30, 2022. https://www.jetbrains.com/idea/

[18] Coursera learn without limits from world-class universities website. Retrieved April 30, 2022. https://www.coursera.org/

[19] Edx the next era of online learning from world's best institutions website. Retrieved April 30, 2022. https://www.edx.org/