Probabilistic Self-Stabilization with Two-State Machines

Michael C. Montgomery and Chakarat Skawratananond

TR TR-PDS-1995-011 June 1995

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Probabilistic Self-Stabilization with Two-State Machines

Michael C. Montgomery Chakarat Skawratananond

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, Texas 78712

June 1, 1995

Abstract

We present and prove the correctness of a probabilistic self-stabilizing algorithm that circulates a token around an asynchronous ring of identical two-state machines. The number of machines in the ring is odd, and communication is unidirectional. If the initial state of the ring has more than one token, execution of the algorithm results probabilistically in convergence to a state with one token. We analyze the expected convergence span, i.e. the average number of moves to stabilize from an abnormal state, for a few restricted cases. We also present simulation results for the worst case initial state of N tokens with N machines which has a move complexity of $O(N^2)$ for small N.

Keywords

Self-stabilization, probabilistic algorithms, distributed systems.

1 Introduction

The concept of self-stabilization was first introduced by Dijkstra [6]. The system he considered was a ring of machines where each machine was granted a privilege, a boolean function of its own state and the state of its neighbors. A central daemon selected one of the privileged machines and the selected machine made its move. A legitimate state of the system was defined to be a state where exactly one machine had a privilege, i.e. mutual exclusion via circulation of a token was desired. The system was defined to be self-stabilizing "if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves" [6]. In addition, the property of *closure* was required; once the system reached a legitimate state, each possible move kept it in a legitimate state.

Self-stabilization has only recently received the attention of many researchers. It has been embraced as unified approach to fault tolerance and attempts have been made to improve upon

Dijkstra's system as well as apply the concept to existing communication protocols and graph algorithms [1, 2, 4, 9, 10, 13, 15, 16, 17]. For Dijkstra's original system, no self-stabilizing algorithm with less than three states per machine has been discovered. Ghosh has designed a system with two states per machine at the expense of a denser topology [8]. A synchronous ring with two-state machines has been demonstrated by Herman [12]. However, this system only stabilizes *probabilistically.* It is this form of self-stabilization that we explore in this paper. Section 2 presents Herman's algorithm and our modification for an asynchronous ring. We then prove the correctness of our algorithm in Section 3. In Section 4, we present our analysis of the expected convergence span, the average number of moves to stabilize from an illegitimate state. Through simulation we compare our algorithms to Dijkstra's 3-state and k-state algorithms in Section 5. We briefly discuss a simple implementation of our system in Section 6, and Section 7 concludes the paper.

2 A Probabilistic Self-Stabilizing System

In Herman's solution with a ring of two-state machines, all machines are identical and execute the algorithm synchronously. A machine is defined to have a token or privilege when its state is the same as its left neighbor. There are N machines with x_i used to represent the state of machine i. The state can be either 0 or 1. The left neighbor of machine 0 is machine $N-1$. At each time step, all machines execute the following statement:

$$\text{if } x_i = x_{i-1} \text{ then } x_i := \gamma_i.$$

γ_i is random variable that satisfies $\Pr(\gamma_i = 1) = p_i$ and $\Pr(\gamma_i = 0) = (1 - p_i)$ where $0 < p_i < 1$. If N is odd, the system converges probabilistically to a state with one token. We assume that the state of a machine can be corrupted, but the code it executes remains intact. Note that the ring is unidirectional, only the state of one neighbor is needed, and the moves of neighboring machines are interfering, a token can be taken away by a move of the left neighbor.

Dijkstra's original solution required asymmetry and more machine states to achieve the stronger property of self-stabilization. The fact that all machines are identical is an attractive property of Herman's solution. Our aim is to examine its application to an asynchronous system. It turns out that the same algorithm will work asynchronously. Any number of concurrent moves can be viewed as a possible time step of the synchronous algorithm. We do make one modification to how a machine determines its new state. The algorithm is as follows:

$$\text{if } x_i = x_{i-1} \text{ then } x_i := \neg x_i \text{ with probability } p_i$$

where $0 < p_i < 1$. Under the assumption that the machines execute at random, the system converges probabilistically to a state with one token. In other words, malicious scheduling of privileged machines is not allowed [15].

Another possibility for a two-state probabilistic algorithm is to define the privileges as in Dijkstra's k-state solution. However, this requires the use of an "exceptional" machine which executes different code, thus destroying the symmetry of the system. Our aim is to design a symmetric system, so we will use the above algorithm with one further change. If we set all $p_i = 1$, the system will still converge probabilistically as long as the machines do not execute in lockstep forever.

The algorithm for each machine is now deterministic; however, the moves by privileged machines are assumed to occur at random resulting in a nondeterministic system. This algorithm is logical because a machine should always pass the token after exercising its privilege in order to achieve the property of fairness.

Algorithm 1

if $x_i = x_{i-1}$ then $x_i := \neg x_i$.

One disadvantage of the algorithm is that the moves of neighboring machines are still interfering. If we allow a machine to retain a token until exercising its privilege, the moves are noninterfering, i.e. a machine cannot lose a token due to a move of its neighbor. We add a boolean variable *have_token* to indicate the presence of a token. Whenever $x_i = x_{i-1}$, *have_token* is set to true. After using a privilege, the left neighbor's state is examined as before to determine if the local state should be changed. The first **if** statement below corresponds to the statement of Algorithm 1 that is executed by a machine wanting to exercise a privilege.

Algorithm 2

if *have_token* then
    have_token := false;
    if $x_i = x_{i-1}$ then $x_i := \neg x_i$;
endif

on $x_i = x_{i-1} \Rightarrow$ *have_token* := true;

3 Proof of Correctness

We now prove the correctness of Algorithm 1. Let $T.x^t$ be the number of tokens in ring state x at time t. If $x_i \neq x_{i-1}$, we say there is a *transition* at machine i. The following lemma (due to Herman [12]) shows that the number of tokens is always odd if N is odd and always even if N is even.

Lemma 1 $T.x^t \bmod 2 = N \bmod 2$.

Proof. With $T.x^t = N$, the lemma is trivially true. For $T.x^t < N$, we represent the global state x as a vector of bits where b and $\neg b$ represent complementary machine states. The vector consists of alternating strings of b and $\neg b$, so the entire state is given by

$$b^{n_1} \neg b^{n_2} \cdots b^{n_{k-1}} \neg b^{n_k}$$

where n_j is the size of each string and k is the number of transitions in the vector. This implies that $T.x^t = N - k$. Since the first and last bits in the vector are not equal, k is even. Thus, we have $T.x^t \bmod 2 = (N-k) \bmod 2 = N \bmod 2$. □

Lemma 1 shows that the number of tokens in the system changes by multiples of two, and N must be odd for a ring state with one token to exist. The following theorem shows that, barring further corruption, the number of tokens in the system is nonincreasing. Time $t+1$ refers to the next state after time t such that one or more privileged machines have concurrently made a move.

Theorem 2 (Safety) $T.x^t \geq T.x^{t+1}$.

Proof. First consider a machine without a token. No move is possible, so the machine cannot generate a new token. A machine with a token can either remain idle or change state. If it remains idle, it may retain its token or lose its token due to a move by its left neighbor. If it changes state, it may pass the token to its right neighbor or remove a token from its right neighbor. In all cases, the number of tokens in the ring cannot increase. □

The next theorem shows that in every state, at least one machine has a privilege.

Theorem 3 (Liveness) *If N is odd, $T.x^t \geq 1$.*

Proof. This follows immediately from Lemma 1. □

The Safety and Liveness Theorems demonstrate that if there is exactly one token in the system, all subsequent states have exactly one token, i.e. closure is satisfied. The following two theorems show that this token is circulated around the ring in a fair manner.

Theorem 4 (Progress) $T.x^t = 1 \wedge x_i^t = x_{i-1}^t \Rightarrow x_{i+1}^{t+1} = x_i^{t+1}$.

Proof. At time t, machine i is the only privileged machine, so $x_{i+1}^t \neq x_i^t$. The only possible move results in $x_i^{t+1} = \neg x_i^t$. Thus, $x_{i+1}^{t+1} = x_i^{t+1}$. □

Theorem 5 (Fairness) *In an infinite sequence of moves with one token in the system, every machine moves infinitely often.*

Proof. From Theorem 4, a move by machine i passes the token to machine $i+1$. As a result, between two moves of machine i, all other machines must move. □

The following two lemmas are the keys to the proof of convergence. The idea of the proof is to characterize the states of the system using Markov chain theory [11].

Lemma 6 *All states x with $T.x = 1$ are positive recurrent.*

Proof. This follows from Theorems 2, 3, and 5. Starting from a state x with $T.x = 1$, the probability of eventual return to x is 1. Furthermore, assuming each machine does not remain indefinitely idle, the mean recurrence time is finite. □

Lemma 7 *All states x with $T.x > 1$ are transient.*

Proof. Let $D.x^t$ be the minimum distance between two tokens in state x at time t. If $D.x^t = 1$, then there exists at least one pair of adjacent tokens. One possible move of the system is that the machine with the left token of the pair changes state while its right neighbor does not change state, thus reducing the number of tokens in the system. So we have $\Pr(T.x^{t+1} < T.x^t) > 0$. From the Safety Theorem, we see that the probability of eventual return to the state x present at time t is strictly less than 1.

Now consider all states x with $D.x^t > 1$. A possible sequence of moves or token passes leads to a state y at time $t+m$ with $D.y^{t+m} = 1$. Since x leads to y with probability strictly greater than 0 and y is transient, x is also transient. □

The previous two lemmas lead immediately to our main theorem.

Theorem 8 (Convergence) $T.x^t > 1 \Rightarrow \Pr(\exists m > 0 : T.x^{t+m} = 1) = 1$.

Proof. Let $N(x)$ be the number of visits to state x starting from x. By definition,

$$\Pr(N(x) = \infty) = \begin{cases} 1 \text{ if } x \text{ is recurrent} \\ 0 \text{ if } x \text{ is transient.} \end{cases}$$

It follows that the number of visits to transient states, states with more than one token, is finite, so the system converges to a state x with $T.x = 1$. □

The correctness of Algorithm 2 follows from the proof for Algorithm 1. The underlying operation of the machine moves has not been modified, so the system still converges probabilistically to a state with one machine i such that $x_i = x_{i-1}$. At this point, any number of the machines may have retained a privilege. Assuming each machine does not hold its privilege indefinitely, the system stabilizes to a state with one privileged machine because new privileges can only be granted by the circulation of the single token.

4 Analysis of the Convergence Span

The proof of convergence was relatively easy. However, analysis of the average convergence span is quite difficult. Our approach is to analyze a few simple cases of Algorithm 1 and then use simulation to obtain further results.

In this and the following section, we assume the existence of a central daemon which randomly selects one machine at a time to execute the algorithm. Obviously, a machine without a token can do nothing. A selection by the central daemon is referred to as a *round*, and a state change by a privileged machine is called a *move*.

The simplest case to consider is a system with an even number of machines starting from a state with two tokens. In this case, the system will converge to a state with zero tokens. Although this system is not useful in practice, it is a good starting point for analysis. The state of the system is determined by the distances between the two tokens. Since the sum of the two distances always equals N, only one distance is needed to specify the current state. Let D equal the distance from token one to token two. The tokens are adjacent when $D = 1$ or $D = N - 1$. From either of these two states, the tokens are destroyed with probability $1/2$. Note that this situation is equivalent to a simple random walk with absorbing barriers. We have the following result.

Theorem 9 *The expected number of moves until a system with an even number of machines and two tokens stabilizes, starting from $D = k$, $1 \leq k \leq N - 1$, is $k(N - k)$.*

Proof. Let S_k be the expected number of moves until stabilization starting from $D = k$. This leads to the following recurrence relation:

$$S_k = \frac{1}{2}(1 + S_{k+1}) + \frac{1}{2}(1 + S_{k-1}) \text{ for } 1 \leq k \leq N - 1$$

which is subject to the boundary conditions $S_0 = S_N = 0$. Solving the recurrence, we find that

$$S_k = k(N - k). \;\square$$

The maximum value occurs at $k = N/2$ for which we have $S_{max} = N^2/4$.

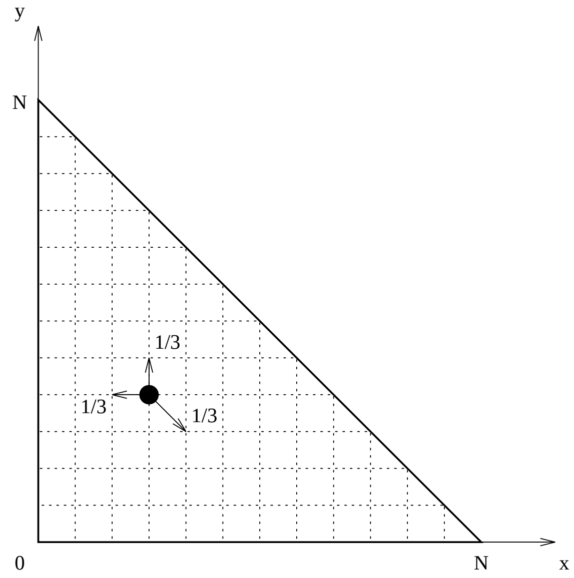

Figure 1: Pictorial representation of convergence from three tokens to one token. The x-coordinate of the particle is the distance from token one to token two. The y-coordinate is the distance from token two to token three. Each of the three possible moves occurs with probability 1/3 and corresponds to a move by one of the three machines holding a token. Convergence occurs when the particle hits one of the barriers.

We now consider a system with an odd number of machines, as required by our algorithms, starting from a state with three tokens. The state of the system is determined by two of three distances between the tokens. This situation is equivalent to a two-dimensional random walk with absorbing barriers in which three moves are possible from each interior position. The motion is constrained to a triangle with base and height equal to N as shown in Figure 1. Let $S_{x,y}$ be the expected number of moves until stabilizing to one token starting from position (x, y) where x is the distance from token one to token two and y is the distance from token two to token three. $S_{x,y}$ satisfies the following recurrence relation:

$$S_{x,y} = \frac{1}{3}(1 + S_{x-1,y}) + \frac{1}{3}(1 + S_{x+1,y-1}) + \frac{1}{3}(1 + S_{x,y+1}).$$

The boundary conditions (corresponding to the perimeter of the triangle) are given by

$$\begin{array}{ll} S_{0,y} = S_{x,0} = 0 & \text{for } \ 0 \leq x, y \leq N \\ S_{x,y} = 0 & \text{for } \ x + y = N, \ 0 \leq x, y \leq N. \end{array}$$

This leads to a system of $(N + 2)(N + 1)/2$ linear equations. We have not found a closed-form solution to the recurrence relation, so we solved the system for values of N less than 85. Starting from three tokens, the position with the highest expectation is $(N/3, N/3)$. The results starting from this position versus the two token case starting from $N/2$ are plotted in Figure 2. As expected, the three token case stabilizes faster. Since the complexity of the expected number of moves for the two token case is $O(N^2)$, the three token case appears to be between $O(N)$ and $O(N^2)$.

The worst case initial state of our system is all machines at 0 or 1. In this case, there are N tokens in the system. Under the assumption of a randomized central daemon, the number of

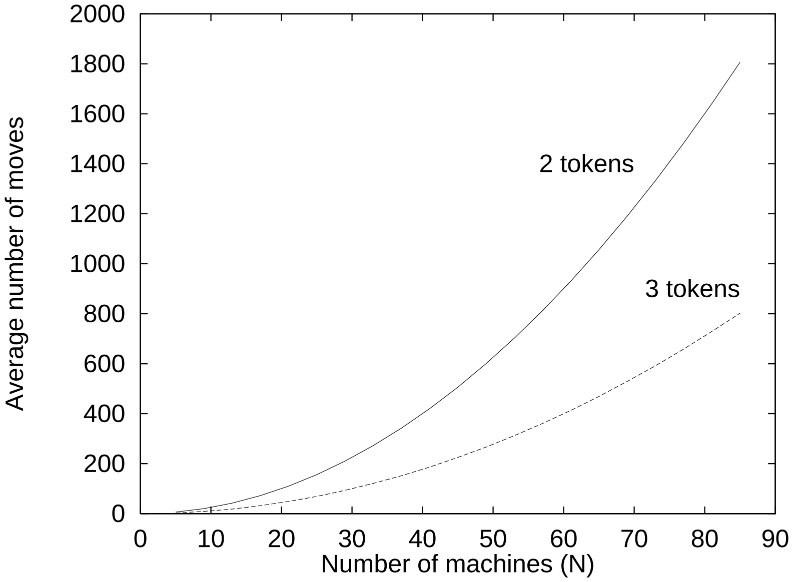

Figure 2: The expected number of moves until stabilization for the two token and three token cases starting from $N/2$ and $(N/3, N/3)$, respectively.

tokens decreases from N to 1 in decrements of 2. Clearly only one move is needed to decrease to $N-2$ tokens, but from there the analysis of the expected number of moves until stabilization becomes extremely difficult. Since the expected number of moves from three tokens to one is less than $O(N^2)$ and $(N-1)/2$ decrements in the number of tokens occur, the expected number of moves from the initial state with N tokens is less than $O(N^3)$.

We first consider a restricted case of this situation where one machine, machine 0, is not allowed to move. Under this assumption, the system satisfies the stronger property of self-stabilization. The lower bound on the number of moves until stabilization is $(N-1)/2$. This corresponds to two tokens being destroyed with every move. The upper bound is $(N-1)N/2$, in which case, machine i moves i times where $0 < i < N$. Thus, Algorithm 1 has a lower bound of $\Omega(N)$ and an upper bound of $O(N^2)$ for this case. Chang et al. have analyzed a similar restricted case for Dijkstra's k-state algorithm and determined that the expected number of moves is $O(N^{1.5})$ [5]. Analysis of this restricted case led us to an alternative algorithm that stabilizes faster than Algorithm 1. When executing the algorithm while holding a token, machine 0 changes state with probability $p_0 \ll 1$. All other machines have $p_i = 1$ as in Algorithm 1. We will refer to this as Algorithm 3. Of course, this makes the system asymmetric, and the token is not circulated fairly in a stable state.

Algorithm 3
Machine 0: if $x_0 = x_{N-1}$ then $x_0 := \neg x_0$ with probability p_0
Machine i, $0 < i < N$: if $x_i = x_{i-1}$ then $x_i := \neg x_i$.

5 Simulation of the Convergence Span

To determine the expected number of moves until stabilization for the worst case initial state with no restrictions, we turn to simulation. We compare Algorithm 1, Algorithm 3 with $p_0 = 0.1$, Dijkstra's

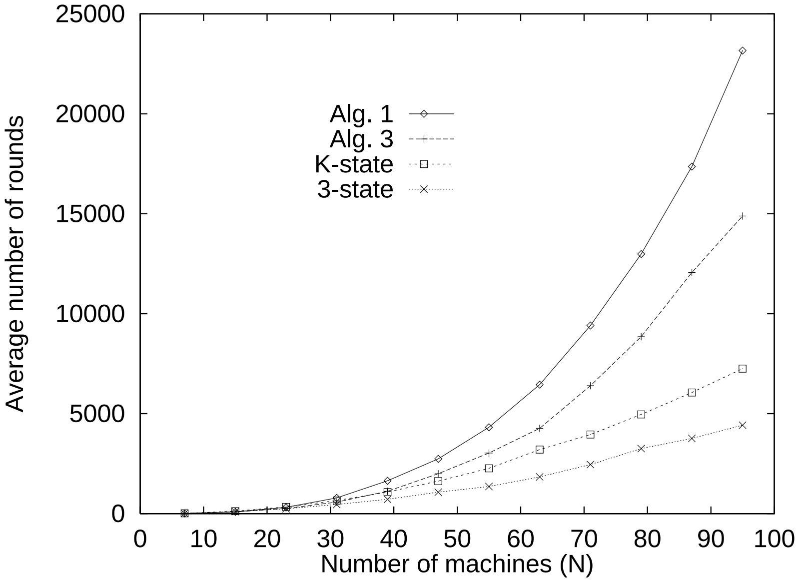

Figure 3: The average number of rounds until stabilization starting from the worst case initial state for Algorithm 1, Algorithm 3 with $p_0 = 1$, Dijkstra's 3-state algorithm, and Dijkstra's k-state algorithm.

3-state algorithm, and Dijkstra's k-state algorithm [6]. The initial state used for our algorithms was $(0, 0, 0, \ldots)$. For the 3-state algorithm we used $(0, 1, 2, 0, 1, 2, \ldots)$, and for the k-state algorithm we used $(0, 1, 2, \ldots, N-2, 0)$. Each of these initial states places a maximum number of tokens in the system. In each simulation, multiple trials starting from the initial state were performed until obtaining a 95% confidence interval within 5% of the mean for both the average number of rounds and the average number of moves until stabilization. The results are shown in Figures 3 and 4 as a function of the system size N. Note that, unlike the other three algorithms, communication in the 3-state algorithm is bidirectional, and the token is reflected by the top and bottom machines instead of circulating around the ring in one direction. This accounts for its faster stabilization. The number of moves required for Algorithm 3 is competitive with the k-state algorithm for these small values of N. The number of rounds though is much greater which implies that it spends a large portion of time with only a few tokens, and, as N grows large, will become worse than the k-state algorithm. For Algorithm 1, the results clearly indicate that we have sacrificed performance to obtain a symmetric algorithm with a small number of states. By comparing Figure 4 with Figure 2, we see that the majority of the moves occur with a small number of tokens in the system. This is the same as the conclusion for Algorithm 3. From the simulation results and the fact that the three token case dominates the convergence span, the average number of moves for Algorithm 1 is $O(N^2)$ for small N.

6 Implementation

Our algorithm affords a simple implementation. Whenever the state of a machine changes or upon recovery from a failure, the machines sends its state to its right neighbor. Therefore, each machine

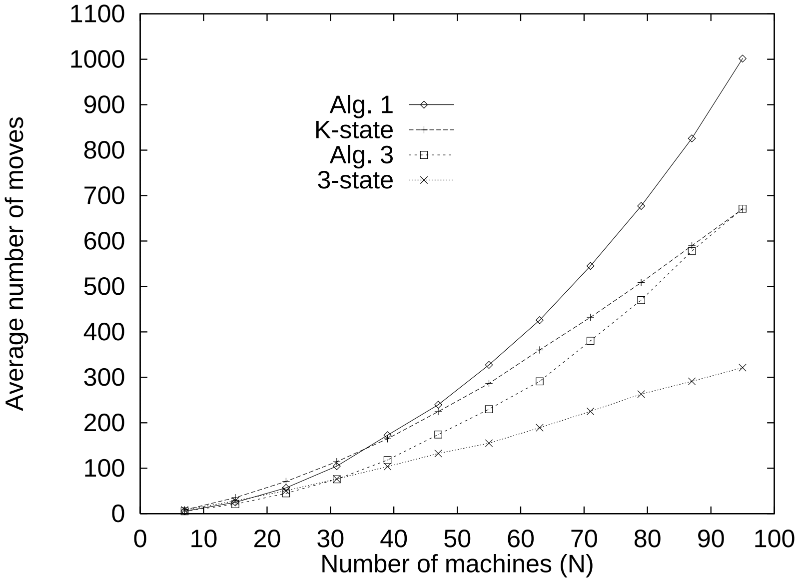

Figure 4: The average number of moves until stabilization starting from the worst case initial state for Algorithm 1, Algorithm 3 with $p_0 = 1$, Dijkstra's 3-state algorithm, and Dijkstra's k-state algorithm.

has access to its local state and the state of its left neighbor. The total number of messages with this algorithm is equal to the total number of moves.

The randomized central daemon of Section 4 can be implemented under the assumption that it operates according to a Poisson process with rate λ [5]. Thus, the time between selections of a randomly chosen machine is exponentially distributed with mean $1/\lambda$. A distributed system where the attempts to move at each machine are independent Poisson processes with rate λ/N is equivalent to this central daemon. This is due to the fact that the sum of independent Poisson processes is also Poisson with a rate equal to the sum of the individual rates. Note that our algorithm does not require a randomized central daemon. It will converge probabilistically as long as machines execute asynchronously and move at random.

7 Conclusion

We have presented and proved the correctness of a probabilistic self-stabilizing algorithm that circulates a token around an asynchronous ring of identical two-state machines. The number of machines in the ring is odd, and communication is unidirectional. If the initial state of the ring has more than one token, execution of the algorithm results probabilistically in convergence to a state with one token. Versions of the algorithm with interfering and noninterfering state changes of neighboring machines were given. An asymmetric version which stabilizes faster was also proposed. It has previously been shown that, under the assumption of a central daemon and deterministic execution, the minimum number of states per machine in a token ring that self-stabilizes within a finite number of moves is three [15]. Our algorithm is deterministic at each machine, but the moves by privileged machines are assumed to occur at random resulting in a nondeterministic system.

We analyzed the expected convergence span, i.e. the average number of moves to stabilize from an abnormal state, for a few restricted cases. Specifically, we obtained a recurrence relation for the expected convergence span from three tokens to one. This system of equations can be solved for a given system size N, but we did not find a closed-form solution. We also presented simulation results for the worst case initial state of N tokens with N machines which has a move complexity of $O(N^2)$ for small N. Dijkstra's 3-state and k-state algorithms stabilize faster than our algorithms at the expense of asymmetry, an increased number of states, and, for the 3-state algorithm, bidirectional communication.

Our conclusion is that probabilistic self-stabilization, a weaker condition than self-stabilization, is useful for transient failure recovery in distributed systems without real-time requirements. By exploiting randomness, the design of a system can be simplified and the state space reduced. There is a need for better convergence analysis of proposed algorithms. In addition, new algorithms which are simple and stabilize more quickly are desirable since the convergence span for current algorithms becomes impractical as N grows large.

References

[1] Y. Afek and G. M. Brown, "Self-stabilization of the alternating-bit protocol," *Proc. of the 8th Symposium on Reliable Distributed Systems*, pp. 80–83, 1989.

[2] B. Awerbuch and R. Ostrovsky, "Memory-efficient and self-stabilizing network reset," *ACM Symposium on Principles of Distributed Computing*, pp. 254–263, Aug. 1994.

[3] G. M. Brown, M. G. Gouda, and C.-L. Wu, "Token systems that self-stabilize," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 845–852, June 1989.

[4] J. E. Burns and J. Pachl, "Uniform self-stabilizing rings," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 2, pp. 330–344, Apr. 1989.

[5] E. J. H. Chang, G. H. Gonnet, and D. Rotem, "On the costs of self-stabilization," *Information Processing Letters*, vol. 24, no. 5, pp. 311–316, Mar. 16, 1987.

[6] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, Nov. 1974.

[7] E. W. Dijkstra, "A belated proof of self-stabilization," *Distributed Computing*, vol. 1, no. 1, pp. 5–6, Jan. 1986.

[8] S. Ghosh, "Binary self-stabilization in distributed systems," *Information Processing Letters*, vol. 40, no. 3, pp. 153–159, Nov. 8, 1991.

[9] S. Ghosh, "An alternative solution to a problem on self-stabilization," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 4, pp. 735–742, Sept. 1993.

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

[11] G. R. Grimmet and D. R. Stirzaker, *Probability and Random Processes*, New York: Oxford University Press, 1992.

[12] T. Herman, "Probabilistic self-stabilization," *Information Processing Letters*, vol. 35, no. 2, pp. 63–67, June 29, 1990.

[13] S.-T. Huang and N.-S. Chen, "A self-stabilizing algorithm for constructing breadth-first trees," *Information Processing Letters*, vol. 41, no. 2, pp. 109–117, Feb. 14, 1992.

[14] J. L. W. Kessels, "An exercise in proving self-stabilization with a variant function," *Information Processing Letters*, vol. 29, no. 1, pp. 39–42, Sept. 15, 1988.

[15] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, no. 1, pp. 45–67, Mar. 1993.

[16] S. Sur and P. K. Srimani, "A self-stabilizing algorithm for coloring bipartite graphs," *Information Sciences*, vol. 69, no. 3, pp. 219–227, 1993.

[17] G. Varghese, "Self-stabilization by counter flushing," *ACM Symposium on Principles of Distributed Computing*, pp. 244–253, Aug. 1994.