

Anonymous Daemon Conversion in Self-stabilizing Algorithms by Randomization in Constant Space

W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and Pradip K. Srimani

School of Computing, Clemson University, Clemson, SC 29634-0974

Abstract. We propose a generalized scheme that can convert any algorithm that self-stabilizes under an unfair central daemon into a randomized one that self-stabilizes under a distributed daemon, using only constant extra space and without IDs. If the original algorithm is anonymous the resulting self-stabilizing algorithm is also anonymous. We provide a detailed complexity analysis that show that the expected slowdown is upper bounded by $O(n^3)$.

1 Introduction

A self-stabilizing algorithm is a distributed algorithm designed to converge to a desired global system state without any external coordination or global system initialization. Each node participates in the distributed algorithm based on local knowledge: its own state and the states of its immediate neighbors. The goal is to achieve some global objective – a predicate defined on the states of all the nodes in the network – based on local actions where individual nodes have no global knowledge about the network. Self-stabilizing algorithms are robust (fault tolerant) in the *optimistic* sense that the distributed system may temporarily behave inconsistently but a return to correct system behavior is guaranteed in finite time while traditional robust distributed algorithms follow a *pessimistic* approach in that it protects against the worst possible scenario which demands an assumption of the upper bound on the number of faults.

A self-stabilizing algorithm is usually written as a collection of production *rules* at each node: each rule specifies a condition and an action. The *condition* is a boolean predicate on the state of the node and the states of its neighbors; the action or *move* is a change in the state of the node executing the action. A node is *privileged* at a particular time if the condition of one or more of its rules is satisfied. Note that a node might stop being privileged if a neighbor moves. We work in the *shared-variable* model in which a node can directly read its neighbors' variables. We restrict attention to undirected, bidirectional links. All computation by a node is completed in one atomic step.

In order to analyze the correctness and time complexity of a self-stabilizing algorithm, a daemon or an execution model (run time environment) is assumed:

the daemon plays the role of both scheduler and adversary. In the literature, there are several daemons, and several possible attributes of those daemons. The *central daemon* (or serial daemon) chooses or *taps* exactly one privileged node to move at each step. In contrast, the *distributed daemon* taps a nonempty subset of the privileged nodes to move at each step. These daemons are considered adversarial. A daemon can be fair or unfair. For a (weakly) *fair daemon*, every node that is continuously privileged is tapped eventually; for an *unfair daemon*, there is no such restriction.

In general, algorithms and protocols are designed (and analyzed) assuming a specific daemon; and, an algorithm designed for one daemon may not work with another daemon in a straightforward way. In order to simplify algorithm development, it would be useful to have mechanisms or procedures to convert an algorithm, designed for one daemon to work with other daemons. These procedures are sometimes called transforms [6]. The concept of daemon can be thought of as two parts: The central daemon promises exclusivity (local mutual exclusion), while the fair daemon promises each processor gets its turn (fairness or clock synchronization). Most of the work on local mutual exclusion, fairness or daemon conversion assumes IDs. This includes the ULME algorithm of [2], the alternator of [4], the conversions of [11] based on the dining philosophers problem, the fairness enhancements of [9], the timestamp-based transforms of [10], and algorithms in the book [3]. Awerbuch et al. [1] provided algorithms for clock synchronization for both anonymous and ID-based networks.

Randomization can be used to ensure local mutual exclusion, and hence to convert a central-daemon algorithm to a distributed-daemon one. One approach is to use randomness to obtain “approximate” IDs (such as a *neighborhood-unique numbering* [5]) for the nodes. However, this requires a stabilization period to establish the node IDs and non-constant additional space. Shukla et al. [12] provided a method using randomness that can be used to convert some specific central-daemon algorithms to run under a distributed daemon.

In this paper, we propose a general algorithm, that converts any arbitrary algorithm (stabilizing under an unfair central daemon) randomization We show that, by using randomness but not to one that self-stabilizes under a distributed daemon. The proposed algorithm has three distinct advantages: (1) it is applicable to *all* self-stabilizing algorithms in central daemon; (2) it does not assume that the nodes in the network have unique *IDs*—it works for *anonymous* algorithms (using node identifiers for reference purpose only). The resulting algorithm is scalable (especially suitable for dynamic networks; departure and arrival of new nodes are most efficient); and (3) the conversion is achieved with a single additional bit at each node (so that if the original algorithm ran in constant space the transformed one does too). The trade-off is a slow-down. We measure the running time of the algorithms in terms of the maximum number of *steps* needed for the algorithm to converge to a stable legitimate state in the worst case. We provide the correctness and convergence analysis of the proposed algorithm and an upperbound of the resulting slowdown.

2 The Conversion Algorithm

Let \mathcal{S} be a given arbitrary self-stabilizing algorithm that works for an unfair, central daemon. We want to design a general conversion algorithm that transforms \mathcal{S} to a new self-stabilizing algorithm \mathcal{S}' , that works for a distributed daemon.

The algorithm/transform is presented as the code for a node i . For the new algorithm \mathcal{S}' , we add to each node i a boolean flag $b(i)$ in addition to the \mathcal{S} -variables (the variables that are used by the algorithm \mathcal{S} at each node i). The design approach is to make this flag $b(i)$ true if the node i is privileged for the underlying algorithm \mathcal{S} and is the only node in its neighborhood that has its b -flag set; when two (or more) adjacent nodes are simultaneously \mathcal{S} -privileged and have their flags set, the nodes randomly determine a new value of their flag bits. A node i can execute the underlying algorithm \mathcal{S} only if it is indeed privileged for \mathcal{S} and is the only node in its neighborhood that has its b -flag set. We define the boolean predicate $p_{\mathcal{S}}(i)$ (in terms of \mathcal{S} -variables at node i) as true iff node i is privileged for algorithm \mathcal{S} in a given system state. The notation $N(i)$ denotes the set of neighbors of node i . The new algorithm \mathcal{S}' is shown as Algorithm 1 [we use Algorithm 1 and Algorithm \mathcal{S}' interchangeably].

Algorithm 1. *Using randomness for exclusivity*

Variables: binary $b(i)$ (and variables needed for \mathcal{S})
BitClear: **if** $b(i) = 1$ **and** not $p_{\mathcal{S}}(i)$
 then set $b(i) = 0$
BitSet: **if** $p_{\mathcal{S}}(i)$ **and** $b(i) = 0$ **and** $\forall j \in N(i) : b(j) = 0$
 then set $b(i) = 1$
BitToss: **if** $p_{\mathcal{S}}(i)$ **and** $b(i) = 1$ **and** $\exists j \in N(i)$ with $b(j) = 1$
 then set $b(i) = \text{RANDOM}$ (toss a fair coin to determine the new value of $b(i)$)
Step: **if** $p_{\mathcal{S}}(i)$ **and** $b(i) = 1$ **and** $\forall j \in N(i) : b(j) = 0$
 then execute one step of \mathcal{S} at i

Note: In Rule BitToss, setting $b(i)$ to RANDOM means tossing a fair-coin to determine the new value of $b(i)$ (either 0 or 1).

2.1 Correctness Analysis

Lemma 1. *Under a distributed daemon, Algorithm 1 (Algorithm \mathcal{S}')*

- achieves local exclusivity for \mathcal{S} , i.e., no two adjacent nodes execute the underlying algorithm \mathcal{S} (executing Rule Step) concurrently;
- cannot terminate while there is an \mathcal{S} -privileged node (i.e., Algorithm 1 terminates only when Algorithm \mathcal{S} terminates).

Proof. – For node i to be able to execute Rule Step, at the point it is tapped it must have its b -bit set, and none of its neighbors can have their b -bit set. Thus, if node i is privileged for Rule Step of Algorithm 1 in a given system state, no neighbor j of node i can also be privileged for Rule Step. Thus two adjacent nodes cannot execute Rule Step simultaneously.

- Assume Algorithm 1 terminates. Then, if a node has its b -bit set, it must be privileged for one of Rules **BitClear**, **BitToss** and **Step** (since their hypotheses are exhaustive). So, when Algorithm 1 terminates, each node i has $b(i) = 0$. But then there cannot be a \mathcal{S} -privileged node, since any such node would be privileged for Rule **BitSet**. So, when algorithm \mathcal{S}' terminates, there is no \mathcal{S} -privileged node.

2.2 Convergence Analysis

Thus it remains to show that, no matter what the distributed daemon does, there is progress on \mathcal{S} . That is, we need to show that the expected time between two consecutive steps of \mathcal{S} is bounded. So, define:

*T is a maximal interval (sequence of steps executed by the distributed daemon) such that Rule **Step** is not executed.*

The steps within T are denoted by an integer variable t , $t \geq 0$. If the daemon is perpetually lucky, T can be infinite; but we now argue that the expected length of T is bounded.

Definition 1. *The set of \mathcal{S} -privileged nodes during the interval T is denoted by P ; by the definition of T , the set P does not change during T . The set of nodes that have their b -bit set at the start of step t is denoted by $B(t)$ and the set of those with their b -bit clear is denoted by $C(t)$ [$B(t) \cup C(t) = V$, V is the set of nodes in the graph].*

Definition 2. *We define a node i of P as **stuck** if $b(i) = 1$ and $b(j) = 0$ for each neighbor j of node i . We define a node i in $C(t) - P$ as **dead** [a dead node cannot make any move while it is dead].*

This terminology is motivated by the following lemma:

Lemma 2. *During the interval T :*

- (a) *a stuck or dead node cannot move;*
- (b) *a node in $B(t) - P$ can move only to become dead.*

Proof. (a) A stuck node is privileged only for Rule **Step** of Algorithm 1 and by definition the Rule **Step** is not executed during the interval T .

(b) Since P does not change during T , the only move a node in $B(t) - P$ can make is to clear its b -bit; this places it in $C(t+1) - P$, i.e., it becomes dead.

Note: A node in $C(t) \cap P (= P - B(t))$ can execute only Rule **BitSet**; by doing so, the node either becomes a stuck node or it enters the set $\$ = P \cap B - stuck$. A node in $B(t) \cap P - stuck$ can execute only Rule **BitToss**: depending on the result of the coin, this puts it in $C(t+1)$ or $B(t+1)$ — the node either becomes stuck, or remains in the set $B(t+1) \cap P - stuck$ or enters the set $P - B(t+1)$. The possible movement of a node from set to set is depicted in Figure 1.

The intuition behind the subsequent analysis is that if the interval T is to continue for a long time, then the daemon must keep on tapping the non-stuck \mathcal{S} -privileged nodes, clearing and setting their bits, without creating stuck nodes. This we formalize using a potential function argument.

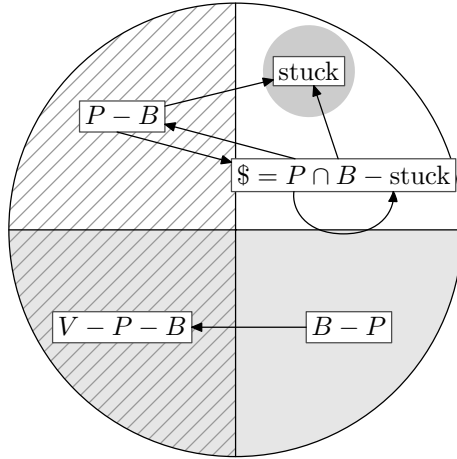


Fig. 1. Set changes without a Step

Intervals Without Stuck Or Dead Nodes Arising

Define a maximal subinterval $T^\#$ of T such that no new stuck or dead node is created.

Observation 3. By Lemma 2, T is divided into at most n such intervals.

We now bound the expected length of $T^\#$.

Definition 3. (a) $H(t)$ is defined to be the subgraph induced by the nodes in the set $B(t)$, and $m(t)$ denotes the number of components of $H(t)$ that contain only nodes of P (S -privileged).

(b) $q(t)$ denotes the number of nodes in $B(t)$ that are S -privileged, i.e., $q(t) = |B(t) \cap P|$.

(c) We define a potential function $\phi(t)$ as follows:

$$\phi(t) = q(t) - (n + 1) \times m(t).$$

It is obvious that $-n^2 - n \leq \phi(t) \leq n$. We will now focus on the potential function. It is sufficient to bound how long before the potential function runs out, since that is a bound on the length of $T^\#$. If it were the case that $\phi(t)$ always decreased, then one would immediately have an $O(n^2)$ bound on the length of $T^\#$. Unfortunately, though $\phi(t)$ mostly decreases, the daemon can get lucky. But we will show that in order for the daemon to get the potential function to increase, the daemon must run the risk that $T^\#$ ends.

Definition 4. Let $A(t)$ denote the subset of nodes tapped by the daemon at time t .

Note that at time t , each node in $A(t)$ is privileged for \mathcal{S}' . We may assume that no node of $A(t)$ is in $B(t) - P$, since that node would become dead, thus terminating $T^\#$.

Observation 4. *We may assume that the subgraph induced by $A(t)$ is connected.*

Proof. Since the daemon is trying to make the interval $T^\#$ as long as possible, one may assume that if one step can be split into two independent steps that are equivalent, the daemon does so. So if the subgraph induced by $A(t)$ were not connected, the daemon would tap the different components in consecutive steps.

If all nodes in $A(t)$ execute Rule **BitToss**, then it is possible for there to be no change (all coins come up 1). However:

Observation 5. *We may assume that every step results in a change in the global state.*

Proof. Recall that the daemon is adversarial. It follows that if $A(t)$ was the correct choice for the daemon the first time, and nothing changed, it is the correct choice the next time. And if nothing changes then, it is the correct choice again. Since the probability of nothing changing is at most $\frac{1}{2}$ (actually it is smaller), the expected number of steps before there is a global change is 2. So, for a constant factor of 2 in the analysis, we may assume that every step results in a change in the global state.

Lemma 6. *If $A(t) \subseteq B(t)$ but not all of a component of $H(t)$, then the potential function ϕ decreases.*

Proof. Every node of $A(t)$ executes Rule **BitToss**. Since $A(t)$ is not all of a component of $H(t)$, $m(t)$ cannot decrease. On the other hand, $q(t)$ decreases unless every node has its coin come up 1; that possibility is taken care of by Observation 5. The result follows.

Lemma 7. *If $A(t) \not\subseteq B(t)$, then the potential function ϕ decreases.*

Proof. Then $A(t)$ contains a node i of $P - B(t)$. The node i has $b(i) = 0$ and such a node is privileged only if all its neighbors have a clear b -bit. Since the subgraph induced by $A(t)$ is connected, it follows that every node in $A(t)$ and all their neighbors have clear bits. Further, in the step, each node in $A(t)$ sets its b -bit, and thus $A(t)$ forms a new component of $H(t+1)$. It follows that $m(t)$ increases while $q(t)$ increases by at most n ; thus ϕ decreases.

If the hypotheses of the above two lemmas do not hold, then $A(t)$ is all of a component of $H(t)$. In this case, all of $A(t)$ execute Rule **BitToss**. Note that $|A(t)| \geq 2$, since otherwise the single tapped node would be stuck.

Lemma 8. *Suppose $A(t)$ is all of a component of $H(t)$. Suppose $|A(t)| = y$ and that z nodes' coin tosses come up so that they remain with bit set. (We assume $z = y$ does not occur, by Observation 5.)*

- (a) *If $2 \leq z < y$, then ϕ decreases.*
- (b) *If $z = 0$, then ϕ increases by less than n .*
- (c) *If $z = 1$, then $T^\#$ ends.*

Proof. Note that $q(t+1) = q(t) - (y - z)$.

(a) Assume $2 \leq z < y$. In this case, no component disappears, and indeed, the component $A(t)$ might split; it follows that $m(t)$ does not decrease while $q(t)$ does decrease. Thus ϕ decreases.

(b) Assume $z = 0$. In this case, $q(t)$ decreases by y , but the component $A(t)$ of $H(t)$ disappears, so that $m(t+1) = m(t) - 1$. It follows that ϕ increases by less than n .

(c) Assume $z = 1$. In this case, the node which remains with bit set becomes stuck, and $T^\#$ ends.

We call the three cases of Lemma 8 Events (a), (b) and (c).

Lemma 9. *The probability of Event (c) is at least twice the probability of Event (b).*

Proof. Since the coin tosses are independent, in any one step, the chance of Event (c) is $y2^{-y}$, while the chance of Event (b) is 2^{-y} . If we adjust for excluding the possibility of $z = y$, both probabilities are divided by $1 - 2^{-y}$, so their ratio remains y . The result follows since $y \geq 2$.

Finally, before we are able to bound the length of $T^\#$, we need to introduce a simple gambling game.

A Gambling Game

Consider a *gambling game* defined as follows. Let ψ_0 and x be positive integers, and α a number between 0 and 1. Consider a person Damon who starts a gambling game with ψ_0 chips. At each step Damon pays 1 chip and has two choices:

- He can choose to not gamble. In this case the step ends.
- He can choose to gamble. In this case he tosses a coin: if the coin comes up heads, he gains $x + 1$ chips; but if it comes up tails, the game ends. The coin comes up heads with probability α .

The game ends when either the coin comes up tails or Damon runs out of money. Damon's goal is to maximize the length of the game.

Lemma 10. *Under optimal strategy by Damon, the expected length of the gambling game is $\psi_0 + x\alpha/(1 - \alpha)$.*

Proof. Suppose that if Damon were to take the coin and toss it repeatedly, then it would come up heads L times before coming up tails. Then, no matter what Damon does, the game will last at most $\psi_0 + Lx$ steps. On the other hand, there is a simple strategy that guarantees the game lasts this long: Damon should wait until having only 1 chip left each time to gamble.

Since L is a geometric random variable, the expected value of L is $\alpha/(1 - \alpha)$. It follows that the expected length of the game is at most $\psi_0 + x\alpha/(1 - \alpha)$, and Damon can achieve this by gambling only when he has 1 chip left.

Lemma 11. *For Algorithm 1, consider a maximal interval $T^\#$ such that Rule Step is not executed and no node becomes dead or stuck. Then the expected length of $T^\#$ is at most $O(n^2)$ steps, no matter the daemon's choices.*

Proof. By Lemmas 6 and 7, the potential function ϕ always decreases unless $A(t)$ is a component of $H(t)$. Call such a choice of $A(t)$ a **component choice**. Say $y = |A(t)|$ and z nodes keep their bit set.

Now, suppose we give the daemon further powers; this can only make the daemon's job easier. We give the daemon the power to partially choose the future with a component choice. Specifically, the daemon can choose whether Event (a) occurs or not; that is, he can choose whether $z \geq 2$ or not. We define a **gamble** as the event that the daemon chooses that z is to be less than 2. (Note that if $y = 2$ then a component choice is automatically a gamble.)

It follows that the expected length of $T^\#$ is at most the expected length of the gambling game for Damon playing optimally, with the following parameters. The total number of coins is $\psi_0 = n^2 + 2n$, the difference between the upper and lower extremes of the potential function. The chance of success α is the conditional probability of Event (b) given that it is not Event (a), which is at most

$$\max_{y \geq 2} 2^{-y} / (2^{-y} + y2^{-y}) = \max_{y \geq 2} 1 / (1 + y) = 1/3.$$

And the increase if lucky is (at most) $x = n$.

So by Lemma 10, the expected length of $T^\#$ is at most $n^2 + O(n)$.

Theorem 1. *Any algorithm \mathcal{S} that self-stabilizes under an unfair central daemon can be converted to a randomized one \mathcal{S}' that self-stabilizes under an unfair distributed daemon, using constant extra space, without IDs, and with at most $O(n^3)$ expected slowdown.*

Proof. Since T is divided into at most n such intervals $T^\#$ (see Observation 3), it follows that the expected length of T is at most $O(n^3)$. This establishes the theorem.

We suspect that the above analysis can be improved to show that the slowdown is at most $O(n^2)$.

2.3 Conclusion

These results reaffirm that in the deterministic ID-based shared-variable model, all daemons are equally powerful. The same result holds in link-registers, since link-registers and shared-variable are equivalent in ID-based networks.

The interesting question is of comparing the power of daemons in deterministic anonymous networks. A distinguished node (root) and link-registers suffice to allow leader election and thus the assigning of IDs, and hence daemons are equally powerful. However, without a root, in both the link-register and uniform shared-variable case, the results of [8,7] and others on rings show that the distributed and central daemon have different powers.

Acknowledgement

This work has been supported by NSF grant # ANI-0218495.

References

1. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: STOC 1993. Proceedings of the 25th Annual ACM Symposium on Theory of Computing, pp. 652–661 (1993)
2. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilizing local mutual exclusion and daemon refinement. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 223–237. Springer, Heidelberg (2000)
3. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
4. Gouda, M.G., Haddix, F.: The alternator. In: Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS 1999), pp. 48–53. IEEE Computer Society, Los Alamitos (1999)
5. Gradinariu, M., Johnen, C.: Self-stabilizing neighborhood unique naming under unfair scheduler. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, pp. 458–465. Springer, Heidelberg (2001)
6. Herman, T.: Models of self-stabilization and sensor networks. In: Das, S.R., Das, S.K. (eds.) IWDC 2003. LNCS, vol. 2918, pp. 205–214. Springer, Heidelberg (2003)
7. Hoepman, J.H.: Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 265–279. Springer, Heidelberg (1994)
8. Israeli, A., Jalfon, M.: Uniform self-stabilizing ring orientation. *Information and Computation* 104, 175–196 (1993)
9. Karaata, M.H.: Self-stabilizing strong fairness under weak fairness. *IEEE Transactions on Parallel and Distributed Systems* 12, 337–345 (2001)
10. Mizuno, M., Kakugawa, H.: A timestamp based transformation of self-stabilizing programs for distributed computing environments. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 304–321. Springer, Heidelberg (1996)
11. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing* 62(5), 766–791 (2002)
12. Shukla, S., Rosenkrantz, D., Ravi, S.: Developing self-stabilizing coloring algorithms via systematic randomization. In: Proceedings of the International Workshop on Parallel Processing, pp. 668–673 (1994)