# Self-Stabilizing l-Exclusion Algorithms

Mitchel Flatebo, Ajoy Kumar Datta, and Anneke A. Schoone

Technical Report RUU-CS-93-01
January 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0024-3275

# Self-Stabilizing l-Exclusion Algorithms

Mitchell Flatebo and Ajoy Kumar Datta
Department of Computer Science, University of Nevada at Las Vegas
Las Vegas, NV 89154

Anneke A. Schoone*
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

**Abstract**

A distributed system consists of a set of loosely connected machines that do not share a global memory. The system is *self-stabilizing* if it can be started in any global state and achieves its consistency all by itself. This also means that the system can deal with *infrequent* errors. This paper presents self-stabilizing $l$-exclusion algorithms. The *l-exclusion problem* is a generalization of the mutual exclusion problem (where $l = 1$). The algorithms presented are generalizations of a self-stabilizing mutual exclusion algorithm by Dijkstra [5]. The first algorithm serves to illustrate the underlying ideas but uses unbounded numbers. The second algorithm is adapted to the use of bounded numbers, while the third algorithm overcomes a disadvantage of the first two algorithms.

**Keywords :** Fault-tolerance, l-exclusion, mutual exclusion, self-stabilization.

## 1 Introduction

A distributed system consists of a set of loosely connected machines that do not share a global memory. Depending on the connectivity and propagation delay in the system, each machine gets a partial view of the global state. The set of global states can be split up into two categories, legal and illegal. A desired property of the system is that regardless of the initial state of the system, legal or illegal, the system automatically converges to a legal state in a finite number of steps. If the system is able to do this, it is called a *self-stabilizing* system. The definition of a legal state depends on the goal of the system, but at least it is stable, i.e., once the system is in a legal state, it will remain so forever.

As a consequence of the definition of a self-stabilizing system, if an error occurs in the system causing it to be put into an illegal state, this can be viewed as an arbitrary initial global state, the system will correct itself and converge to a legal state in a finite amount of time. However, if we take transient failures into account, a legal state is not stable any

*The research of this author was supported partially by the ESPRIT Basic Research Action No. 7141 (project ALCOM II: *Algorithms and Complexity*), and partially by the Netherlands Organization for Scientific Research (NWO) under contract NF 62-376 (NFI project ALADDIN: *Algorithmic Aspects of Parallel and Distributed Systems*).

more. Unfortunately, this means that in general it is not possible to "observe" whether the system is in a legal or an illegal state. Lin and Simon [12] argue that one should add observation algorithms to self-stabilizing systems to detect whether stabilization has occurred or not. Due to the "instability" of the legal states, it might at most seem possible to detect whether stabilization had occurred and not whether a new failure has disrupted this. According to Katz and Perry [11], even this is impossible.

In 1974, Dijkstra introduced the property of self-stabilization in distributed systems and applied it to algorithms for mutual exclusion [5]. In the mutual exclusion problem, there is an activity, entering a critical section of code, that only one process can do at a time. The *$l$-exclusion problem* is a generalization of this standard problem. The $l$-exclusion problem is defined in the same way except that $l$ processes are now allowed to perform this activity, or enter the critical section at the same time. An example of a practical application is given in [1]: Each process controls a device which from time to time has to enter a mode of high electrical power consumption. The main circuit breaker stands at most $l$ devices at high electrical power consumption. If we allow a high consumption only inside a critical section, $l$-exclusion prevents the burn-out of the circuit breaker.

Algorithms for $l$-exclusion are given in [1, 7, 9, 10, 13]. The problem was first defined and solved by Fischer, Lynch, Burns, and Borodin in a generalized test and set model [9]. The $l$-exclusion algorithms in [7, 10] used a strong read-modify-write primitive. Bounded solutions to the $l$-exclusion problem using only shared memory were given in [1, 7, 13]. The algorithm in [1] solves the problem without using powerful read-modify-write synchronization primitives, however, it is not self-stabilizing.

This paper presents self-stabilizing $l$-exclusion algorithms. The algorithms presented in [13] are simple. However, each algorithm assumes that each process can read variables from every process. This means that each process can read the state of every other process in the system. The algorithms presented in this paper only assume that each process can read the state of its left neighbor in a ring topology. As in Dijkstra's algorithm, this is used to determine whether a process can enter its critical section. Upon returning, it then changes its state such that its right neighbor can enter its critical section. By a slight change in the algorithm we accomplish that this can happen at $l$ processes around the ring simultaneously, thus achieving $l$-exclusion. We first demonstrate a version of the algorithm with unbounded numbers, as the analysis of the version with bounded numbers (the second algorithm) is somewhat intricate. A disadvantage of the first two algorithms is that it is possible that instead of $l$ processes, only one process may enter its critical section, thus in effect, reducing the algorithm to a mutual exclusion algorithm. We remedy this in the third algorithm.

Section 2 describes the problem more formally and gives the notation and the model that will be used in the paper. Section 3 gives an unbounded algorithm and its correctness proof, assuming the presence of a central demon and a distributed demon. Section 4 presents a bounded algorithm for $l$-exclusion and a correctness proof. Furthermore, the value of the bound used in the algorithm is discussed and examples are given to further explain the algorithm in relation to the bound used. Section 5 gives an extended algorithm that remedies a problem of the first two algorithms, and compares the two different algorithms (bounded and extended). Finally, Section 6 gives the results and conclusions of the paper.

# 2 Preliminaries

## 2.1 The Problem

**$l$-Exclusion.** Assume that the program of the processes in a distributed system contains a piece of code called the *critical section*. The problem of $l$-exclusion is to ensure that at most $l$ processes are in their critical section at the same time. The problem is an extension of the mutual exclusion problem (when $l = 1$). As our algorithms are generalizations of one of Dijkstra's solutions ([5]) for the mutual exclusion problem, we will adopt his terminology. As it does not matter for the problem $l$-exclusion what happens inside the critical section, we do not discuss details of when a process enters and when it returns from the critical section but just record the process' ability to do so. In Dijkstra's terms: "the process has a *privilege*". In order to pass a privilege from one process to another, a process can make a *move*. Only privileged processes can make a move. For any solution to the problem of $l$-exclusion,, we require the following properties:

*$l$-Exclusion:* At any time, at most $l$ processes have a privilege.

*Fairness:* Every process moves infinitely often.

It is clear from the definition of $l$-exclusion, that if the number of processes is $n$ and $l \geq n$, no algorithm is necessary.

**Self-Stabilization.** In a distributed system without message passing, the global state is defined as the product of all local states. For the purpose of a self-stabilizing algorithm, the set of all global states $G$ is divided into two sets, the set $L$ of all *legal* states, and the remainder, the set of all illegal states. An algorithm is called *self-stabilizing* if the following two conditions hold: first, starting from an arbitrary global state, it reaches a legal state within a finite amount of time. Second, any step taken while it is in a legal state, leaves it in a legal state. Hence, the required properties for a self-stabilizing $l$-exclusion algorithm are:

*SS-$l$-exclusion:* In a legal state, at most $l$ processes have a privilege.

*Stability:* If a state is legal, then a move results in a legal state.

*SS-Fairness:* If a state is legal, then within a finite number of moves, every process makes a move.

*Convergence:* After a finite number of moves, the state is legal.

For a traditional algorithm the initial state would be a legal state, and all reachable states would be legal states. The illegal states would not be reachable for a traditional algorithm because by assumption the algorithm would not start in such a state.

Note that the set $L$ in general is not equal to the set of global states that correspond to the notion that at most $l$ processes have a privilege. The legal states might have an additional property that ensures that after a move, there are still only $l$ processes with a privilege (Stability Property).

## 2.2 Programming Notation and Execution Models

**Topology.** The topology used in this paper is a ring. For simplicity, it is assumed that only one process runs on a machine, and they are numbered from 0 to $n-1$, counterclockwise. Thus, each machine has two neighbors which we call the left and right neighbors. For a machine $i$ ($0 \leq i \leq n-1$), the left neighbor is machine $i-1$ and the right neighbor is $i+1$, where indices are modulo $n$.

**Programming Notation.** All processes except one (machine 0) run the same program. Such an algorithm is called *non-uniform*. It has been shown that deterministic uniform algorithms for mutual exclusion do not exist for general $n$ [6]. The program for a process has the form :

**begin**
< *action* >
< *action* >
⋮
< *action* >
**end**

Each action has the form :

< *guard* > $\longrightarrow$ < *assignment statement* >

A *guard* is a boolean expression over the variables that a process can read. A process can read its own variables, but also the local variables of its neighbors. In the algorithms that we present in this paper, a process reads only its left neighbor's variables (in addition to its own). The assignment statement updates local variables. Variables are subscripted by the machine number of their owner. Thus, if the processes record their state in $S$, machine $i$ can write $S_i$ and machines $i$ and $i+1$ can read $S_i$. If some process has an action whose guard is true (i.e., it is enabled), then the action may be performed, that is: the assignment statement may be executed. In terms of Dijkstra [5]: then that process has a privilege and may make a move.

**Execution Models.** Usually it is assumed that actions are executed atomically, one at a time. Dijkstra [5] described this model by a *central demon*. If a central demon is assumed to be present, the central demon chooses one process from the set of privileged processes to make a move. No assumptions are made about this choice. However, it is not necessary to make this assumption, and we can allow processes to move simultaneously. Dijkstra called this a *distributed demon*. If a distributed demon is present, at any point in time, any subset of the set of privileged processes can move at this time. As the interleaving of the reads (in the guards) and the writes (in the assignment statements) can be different now, the behavior of the algorithm can differ also. The central demon is much more restrictive, hence it is easier to verify the algorithm. However, Burns et al. [3] developed a theory relating the correctness of an algorithm in the presence of a distributed demon to the correctness of that algorithm in the presence of a central demon.

## 3 Unbounded l-Exclusion Algorithm

The idea for the self-stabilizing mutual exclusion algorithm of Dijkstra [5] is the following. Each process has one local variable, an integer state variable. For all machines except 0, a process has a privilege if its state variable is *unequal* to its left neighbor's value. Conversely, machine 0 has a privilege if its value is *equal* to that of its left neighbor (machine $n-1$). Note that as a consequence there is always at least one machine with a privilege. Doing a move amounts to passing the privilege to the right. For a machine other than 0, this means setting its state variable to the value of its left neighbor, while for machine 0, this means making its state variable unequal to that of machine $n-1$. The latter is done by increasing machine 0's value by one.

Now, consider what global states correspond to only one machine having a privilege. First, all state values could be equal, and machine 0 has the only privilege. Second, state variables could contain one of two different values. If these values would alternate, say, in neighboring machines, all these machines would have a privilege. Hence, the same value must occur in consecutive machines which means that there now are two machines with a state value different from that of its left neighbor. Thus, one of these machines must be machine 0 which does not have a privilege in that case.

Starting from an arbitrary global state, stabilization is achieved as follows. Machine 0 will increase its state value several times until it is equal to the maximum of all values present in the system. Then, it can move once when machine $n-1$ attains this value too, but then, it is disabled until all other values have disappeared from the system by being passed to the right until overwritten at machine $n-1$. Because machine 0 increases its value by 1, the state values in the system differ by at most one after stabilization.

To implement $l$-exclusion, we demand that state values differ by at most $l$. We change only the program for machine 0, and at that only the guard: we say machine 0 has a privilege if its state value is within $l$ of that of its left neighbor (machine $n-1$), instead of being within 1 (i.e., equal). More formally, we have the following.

Each process has one local variable, the state variable $S$ which is an integer. The value of this integer is not bounded. For a process which runs on machine $i$, the *action* can involve the changing of only this variable $S_i$ because this is the only variable owned by the process running on machine $i$. The guard involves the variable of the process running on the left neighbor machine, i.e., $S_{i-1}$ ($S_{n-1}$ for machine 0).

{Algorithm 1: Unbounded $l$-exclusion algorithm}
{for machine 0}
**begin**
$(|S_0 - S_{n-1}| < l) \longrightarrow S_0 := S_0 + 1$
**end**
{for other machines, $i$}
**begin**
$(S_i \neq S_{i-1}) \longrightarrow S_i := S_{i-1}$
**end**

The set of global states $G_1$ for this algorithm is the product space of the $n$ state variables. Thus, for a global state $g \in G_1$ where $g = (g_0, g_1, \ldots, g_{n-1})$, $g_i$ represents the value of the variable $S_i$. The set of legal states $L_1$ is formed by a non-increasing sequence of

state values all within $l$ of each other. This condition ensures that the number of privileges is less than or equal to $l$. Thus:

$G_1 = \mathbb{Z}^n$,

$L_1 = \{g \in G_1 \mid g_0 \geq g_1 \geq ... \geq g_{n-1} \wedge g_0 - g_{n-1} \leq l\}$

## 3.1 Proof for a Central Demon

In this model, we can assume that there is only one move at a time. The line of the proof is as follows. There can be no deadlock, hence there are machines that move. Each machine can only move finitely many times before another machine has to move, so all machines must move eventually (thus, the property of *SS-Fairness* holds). When all machines move, a structure begins to evolve that converges to a legal state (the property of *Convergence*). Once the system is in a legal state, it stays legal, as the legal states are stable (*Stability* property). Thus, the algorithm stabilizes. Moreover, a legal state indeed implies that there are at most $l$ privileges in the system (*SS-l-exclusion*).

**Lemma 3.1** *Starting from an arbitrary state, machine 0 eventually makes a move.*

*Proof:* First note that it is not possible that no machine holds a privilege: if machines 1, 2 up to $n-1$ cannot move, this implies that $S_0, S_1, ...S_{n-1}$ are all equal, hence machine 0 can move. Assume machine 0 does not make a move (otherwise the Lemma holds). All machines examine the state of their left neighbor for the privilege. So, a move by any machine can only cause a privilege to be passed to the right. Thus, all privileges propagate to the right. Represent the system state by a binary number with $n-1$ bits (machine 1 is the most significant bit, machine $n-1$ is the least significant bit). Since machine 0 does not make a move by assumption, it is not included in the binary number. The bit of machine $i$ is 1 if it holds a privilege, and 0 otherwise. If machine $i$ makes a move, its bit changes to a 0 and the bit to the right (machine $i+1$), may become a 0 or a 1. In either case, the value of the binary number has decreased. Hence, the binary number decreases in value after each move. This process is bounded below by 0. So, eventually the system will look like 00...00. At this point, all machines have the same state, and machine 0 is the only privileged machine. Now, machine 0 must make a move. □

**Lemma 3.2** *Between $2l$ moves by machine 0, machine $n-1$ must make at least one move.*

*Proof:* The distance between the states of machines 0 and $n-1$ must be less than $l$ to enable machine 0. Assume the distance is $l-1$ which is the worst case. By Lemma 3.1, machine 0 will make an infinite number of moves in an infinite execution. After one move by machine 0 (assuming $S_{n-1} > S_0$ which is the worst case because machine 0's move causes the distance to be less), the distance will be $l-(1+1)$. After the second move, the distance is $l-(2+1)$. After $l-1$ moves, the distance is $l-(l-1+1)=0$. If machine 0 moves again, the distance will be one. After another $l-1$ moves the distance will be $l$, and machine 0 can not make another move until machine $n-1$ moves. Lemma 3.1 says machine 0 must eventually move which means machine $n-1$ must move. Thus, between $(l-1)+l+1=2l$ moves by machine 0, machine $n-1$ must move at least once. □

**Lemma 3.3** *Starting from an arbitrary state, all machines must eventually move.*

*Proof:* Lemma 3.2 guarantees that machine $n-1$ will eventually move. After this move, $S_{n-1} = S_{n-2}$. By Lemma 3.2, machine $n-1$ will move again eventually. But in order to do this, machine $n-2$ must move to satisfy the condition $S_{n-2} \neq S_{n-1}$. So, between two moves by machine $n-1$, machine $n-2$ must move at least once. In general, for machine $i$ $(0 < i < n-1)$, between two moves by machine $i+1$, machine $i$ must move at least once. Lemmas 3.1 and 3.2 show that machines 0 and $n-1$ must also move eventually. So, starting from an arbitrary state, all machines will eventually move. □

**Lemma 3.4** *After a finite number of moves, the system will be in a state so that for all machines $i$ and $j$ : $S_i \geq S_j$ if $i < j$.*

*Proof:* All machines change their states to be the same as their left neighbor's state except machine 0. Machine 0 increments it state. Assume that the highest state number among the processes in the system is $h$. By Lemma 3.1, machine 0 must move. After $h - S_0$ moves, machine 0 will have the highest state number in the system since $S_0 = h$ and all machines copy the states of their left neighbor. From this point on, machine 0 will have the highest state number in the system (there may be other processes in the system that also have the highest value). Hence, $\forall j : S_0 \geq S_j$ if $0 < j$. After this occurs, any moves by machine 0 will just increase its state number. Thus, these inequalities will continue to hold, i.e., they form a stable property of the system. Machine 1 will eventually move (Lemma 3.3); it will set its state to be the same as machine 0 and will have a state number bigger than or equal to all machines to the right. Now, we have for $i = 0, 1$, that $\forall j : S_i \geq S_j$ if $i < j$, and this property is also stable. After this occurs, machine 2 is guaranteed to move, and then it will have a state number bigger than or equal to all the machines to the right. This process is continued until $S_i \geq S_j$ if $i < j$, for all $i$ and $j$, and this situation will endure forever. □

**Theorem 3.5** *Algorithm 1 stabilizes in the presence of a central demon.*

*Proof:* By Lemma 3.4, eventually, $S_i \geq S_j$ if $i < j$, for all machines $i$ and $j$. Once the system is in this state, it will remain in this state. When the system is in this state, machine 0 will eventually move by Lemma 3.1. So, $(S_0 - S_{n-1}) < l$ (the absolute value is not needed any more). This means that there are at most $l$ distinct numbers. So, there are at most $l$ privileges including machine 0. The privileged machines are those that have a state not equal to its left neighbor's state for machines 1, 2, ..., $n-1$, and machine 0 has a privilege if $S_0 - S_{n-1} < l$. A move by machine 0 will insert a new number in the system. But there can never be more than $l+1$ distinct numbers in the system. If there are $l+1$ distinct numbers, there are only $l$ privileges (one privilege for each number different from $S_0$ and machine 0 is not privileged). Machine 0 cannot make another move until machine $n-1$ makes a move which results in a different value of $S_{n-1}$. Once machine $n-1$ makes such a move, the number of distinct state numbers in the system is decreased. Hence, there are never more than $l+1$ numbers. The definition of $L_1$ is satisfied, and there are never more than $l$ privileges in the system. The system is stabilized since this state is stable. □

## 3.2 Proof for a Distributed Demon

The previous section proved the correctness of the algorithm in the presence of a central demon. This section will show the correctness of the algorithm in the presence of a

distributed demon, thus a number of machines can move at the same time. The line of the proof is the same as for the central demon's case.

**Lemma 3.6** *Starting from an arbitrary state, machine 0 eventually makes a move.*

*Proof:* In the model of a distributed demon, if a machine continually enjoys a privilege, it will eventually make a move. All machines examine their left neighbors which means all privileges propagate to the right. If machine 0 makes a move, the Lemma holds. Assume machine 0 does not make a move. Let machine $i$ be the lowest numbered machine that has a privilege. Machine $i-1$ can only move if machine $i-2$ moves, ..., machine 1 can only move if machine 0 moves. If machine 0 does not make a move, machine $i$ will continually have a privilege. Machine $i$ must eventually make a move. Now, the lowest numbered machine with a privilege has a number greater than $i$. This process can be repeated until machines 1, 2, ..., $n-1$ do not have privileges. At this point machine 0 has the only privilege and must eventually move. □

**Lemma 3.7** *Between $2l$ moves by machine 0, machine $n-1$ must make at least one move.*

*Proof:* The proof is similar to Lemma 3.2. The distributed demon does not play a role because machine 0 increments its state. Once it increments its state so that $S_0 - S_{n-1} = l$ (this occurs after machine 0 moves $2l-1$ times in the worst case), machine $n-1$ must move before machine 0 can move again. Lemma 3.6 guarantees that machine 0 eventually moves. Thus, between $2l-1+1 = 2l$ moves by machine 0, machine $n-1$ must move at least once. □

**Lemma 3.8** *Starting from an arbitrary state, all machines must eventually move.*

*Proof:* The proof is similar to Lemma 3.3 and follows from Lemmas 3.6 and 3.7. □

**Lemma 3.9** *After a finite number of moves, the system will be in a state such that for all machines $i$ and $j$ : $S_i \geq S_j$ if $i < j$.*

*Proof:* As in Lemma 3.7, the proof of this Lemma is very similar to Lemma 3.4. Eventually machine 0 will have the highest state number. After that, large state values will propagate to the right which yields the conclusion of the Lemma (the state values are in a non-increasing sequence). □

**Theorem 3.10** *Algorithm 1 stabilizes in the presence of a distributed demon.*

*Proof:* This theorem follows from Lemmas 3.6 and 3.9. Once there is a non-increasing sequence of system states (Lemma 3.9) and machine 0 makes a move, there can be at most $l+1$ distinct state numbers in the system. This implies that there are at most $l$ privileges in the system. If there are $l+1$ distinct numbers, machine 0 will not be privileged, and there is one machine privileged for each distinct number that is strictly less than the maximum number (the lowest numbered machine having that number). If there are less than $l+1$ distinct numbers, there are at most $l$ privileges in the system (one for each number). The system is stabilized since this state is stable. □

**Observation:** *The main reason the algorithm also works for a distributed demon is because machine 0 increments its state. This avoids any cycles of illegal states caused by multiple machines moving at the same time.*

# 4 Bounded l-Exclusion Algorithm

The major drawback of the previous algorithm is the fact that the state values are unbounded. Machine 0 increments its state when it makes a move. So, the state variable, $S_i$ for machine $i$, is unbounded. This restriction is not needed. The following algorithm bounds the state variables between values of 0 and $k-1$ by using *mod* $k$. How the parameter $k$ of the algorithm is chosen depends on the value of $l$, the number of machines $n$, and the model used (central versus distributed demon). We will discuss this issue in Section 4.2. The corresponding lemmas will be proved for the algorithm to verify the correctness. The verification is more intricate which is why the unbounded variables were used previously.

{Algorithm 2: Bounded form of Algorithm 1}
{for machine 0}
**begin**
$((S_0 - S_{n-1}) \ mod \ k < l) \longrightarrow S_0 := (S_0 + 1) \ mod \ k$
**end**
{for other machines, $i$}
**begin**
$(S_i \neq S_{i-1}) \longrightarrow S_i := S_{i-1}$
**end**

Because the notion of "greater or equal" is not well defined in the cyclic ring $\mathbb{Z}_k$, we have to take some care in defining the set of legal states for this algorithm. We still want, as in $L_1$, that the values of the $S_i$ are non-increasing and remain close to each other, i.e., within a distance of $l$. In $L_2$, we formalize this by demanding that there is at most one processor position $j$, the *modulo jump*, such that the condition in $L_1$ holds if we add $k$ to the state variables of the processors smaller than $j$. Then,

$G_2 = (\mathbb{Z}_k)^n$

$L_2 = (L_1 \cap G_2) \cup \{g \in G_2 \mid \ g_0 + k - g_{n-1} \leq l \ \wedge \ \exists j : 0 < j \leq n-1 :$
$g_0 + k \geq g_1 + k \geq \ldots \geq g_{j-1} + k \geq g_j \geq g_{j+1} \geq \cdots \geq g_{n-1}\}$

## 4.1 Proof for a Central Demon

It is clear that the correctness of Algorithm 2 depends on the value used for $k$. Hence, we will derive bounds on the value of $k$ in the process of proving the algorithm correct.

**Lemma 4.1** *Starting from an arbitrary state, machine 0 eventually makes a move.*

*Proof:* This lemma has the same proof as Lemma 3.1, as the latter did not depend on the unboundedness. □

**Lemma 4.2** *Between $l+1$ moves by machine 0, machine $n-1$ must make at least one move.*

*Proof:* First note that for arbitrary $x$ and for values of $k \leq l$, $x \ mod \ k < l$. This would mean that machine 0 always has a privilege. Thus, the lemma does not hold if $k \leq l$ and we demand $k > l$. Assume machine $n-1$ does not make a move. Then, the value of $S_{n-1}$ remains the same during all $l+1$ moves of machine 0. Let the value of $S_0$ be $p$ before

the first of these $l+1$ moves. Then, $(p - S_{n-1})\ mod\ k < l$, because machine 0 had a privilege. Likewise, before the next moves, $(S_0 - S_{n-1})\ mod\ k = (p - S_{n-1} + 1)\ mod\ k < l$, $(p + S_{n-1} + 2)\ mod\ k < l$, ..., $(p + S_{n-1} + l)\ mod\ k < l$. However, this is a contradiction for values of $k$ such that $k \geq l + 1$. Thus, machine $n-1$ must make at least one move. □

**Lemma 4.3** *Given an arbitrary global state, eventually, every machine will make a move.*

*Proof:* Similar to Lemma 3.3. □

**Lemma 4.4** *After a finite number of moves, the system will be in a legal state.*

*Proof:* This is similar to the proof of Lemma 3.4, and an extension of the proof of Dijkstra [6] for his corresponding algorithm for mutual exclusion. (Figure 1 contains an example

| state of machine 0 | state of machine 1 | state of machine 2 | state of machine 3 | state of machine 4 | privileged machines |
|---|---|---|---|---|---|
| 5 | 4 | 2 | 0 | 5 | 0,1,2,3,4 |
| 6 | 4 | 2 | 0 | 5 | 0,1,2,3,4 |
| 0 | 4 | 2 | 0 | 5 | 1,2,3,4 |
| 0 | 4 | 2 | 0 | 0 | 0,1,2,3 |
| 0 | 4 | 2 | 2 | 0 | 0,1,2,4 |
| 0 | 4 | 4 | 2 | 0 | 0,1,3,4 |
| **0** | **0** | 4 | 2 | 0 | 0,2,3,4 |
| **2** | **2** | **0** | 4 | 2 | 0,2,3,4 |
| **4** | **4** | **2** | **0** | 4 | 0,2,3,4 |
| **5** | **4** | **2** | **0** | 4 | 0,1,2,3,4 |
| **6** | **4** | **2** | **0** | 4 | 1,2,3,4 |
| **6** | **4** | **2** | **0** | **0** | 1,2,3 |
| **6** | **4** | **2** | **2** | **0** | 1,2,4 |
| **6** | **4** | **4** | **2** | **0** | 1,3,4 |
| **6** | **6** | **4** | **2** | **0** | 2,3,4 |
| **6** | **6** | **4** | **2** | **2** | 2,3 |
| **6** | **6** | **4** | **4** | **2** | 2,4 |
| **6** | **6** | **6** | **4** | **2** | 3,4 |
| **6** | **6** | **6** | **4** | **4** | 3 |

In the second and third "rounds", not all moves are shown. States indicated in bold are colored blue according to the proof of Lemma 4.4.

Figure 1: Algorithm 2 using 7 states stabilizes for a central demon.

of a possible execution for $n = 5$, $l = 2$, and $k = 7$.) According to Lemma 4.1, machine 0 eventually moves. According to Lemma 4.3, machine 1 will also eventually move after machine 0. As a result, the states of machines 0 and 1 will be equal. Without loss of generality, assume that the value of these states is 0. Color these two states blue, and

color the states of all other machines white. From then on, after each move, color the state blue if it is obtained by copying a blue state. Color all new states of machine 0 blue too. Let $h$ be the number of moves by machine 0 while the state of machine $n-1$ is white, after the move that produced the first blue state. The number of different values of white states when machine 1 produces its first blue state, is at most $n-2$. By Lemma 4.2, for every $l$ moves by machine 0, machine $n-1$ must move once. This move results in a different value for $S_{n-1}$ than before the move. Hence, after at most $n-2$ (the number of different white states) moves by machine $n-1$ the state $S_{n-1}$ becomes blue. Thus, $h \leq (n-2)l$. If $k > (n-2)l$, then the $h$ moves of machine 0 have produced the blue states from 1 through $h$, apart from the original blue 0, and the blue states of machines 0, 1, ...,$n-1$ form a sequence of non-increasing values. The reason that this sequence is non-increasing, is the same as in the proof of Lemma 3.4. At the next move of machine 0, $(S_0 - S_{n-1})\ mod\ k < l$, and since $S_{n-1}$ is blue, it has a value between 0 and $h$. As $S_0 = h$, $(S_0 - S_{n-1})\ mod\ k = h - S_{n-1}$, and thus $S_0 - S_{n-1} < l$. By definition of $L_2$ this is a legal state. □

**Theorem 4.5** *Algorithm 2 stabilizes in the presence of a central demon.*

*Proof:* This is similar to Theorem 3.5. Once the system is in a legal state, there are at most $l+1$ distinct numbers. There remains to be shown that this implies that there are at most $l$ privileges in the system, and that this state is stable.

If there are $l+1$ distinct numbers, machine 0 can not have a privilege and there can be at most $l$ other privileges. If there are less than $l+1$ distinct numbers, machine 0 can have a privilege, and there will at be at most $l-1$ privileges for machines 1,2,...,$n-1$ (one privilege for each number besides $S_0$). So, the number of privileges is at most $l$.

Let the system be in a state in $L_2$. Then, a move by a machine other than 0 and $j$, the parameter in the definition of $L_2$ (the "modulo jump"), clearly will leave the system in $L_2$. A move by machine $j$ will cause the modulo jump at $j$ to be moved one to the right, so this parameter $j$ will be increased by one (or disappear if $j$ equaled $n-1$). In case machine 0 moves and $S_0$ was smaller than $k-1$, the inequalities will still hold, because machine 0's value will have increased. As before the move $(S_0 - S_{n-1})\ mod\ k < l$, afterwards $(S_0 - S_{n-1})\ mod\ k \leq l$, hence the last condition in $L_2$ still holds. If machine 0 moves while $S_0$ was $k-1$, a new modulo jump is introduced. However, there could have been no modulo jump in the system before, as this would imply that $k-1+k-S_{n-1} \leq l$ which is not possible as $S_{n-1} < k$ and $l < k$. Hence, before the move $k-1-S_{n-1} < l$ and afterwards $0+k-S_{n-1} \leq l$, and the parameter $j$ in the definition is 1. Hence, a state in $L_2$ is moved to a state in $L_2$ and the system is stabilized. □

## 4.2 Value of the Bound

The algorithm works similarly to the unbounded algorithm. All that is changed is the definition of non-increasing sequence. In this algorithm, $k$ states are used to achieve $l$-exclusion.

**Corollary 4.6** *In the presence of a central demon* $(n-2)l+1$ *states are sufficient for* $n \geq 3$ *for Algorithm 2.*

*Proof:* The bounds derived for $k$ during the proof are $k \geq l+1$ (Lemma 4.2) and $k > (n-2)l$ (Lemma 4.4). Hence, the proof above is valid for $k \geq (n-2)l+1$ if $n > 2$, and for $n = 2$ we have a bound of $k \geq l+1$. (For values of $l \geq n$, no algorithm is needed.) □

In [5] Dijkstra claims a bound of $K \geq N$ in the presence of a central demon, but he uses $N+1$ processes, thus $N+1 = n$. Hence, Dijkstra's claim reduces to $K \geq n-1$ while we also have (taking $l = 1$) $k \geq n-2+1 = n-1$.

**A lower bound.** We will first show that the bound derived above is exact.

**Lemma 4.7** *Algorithm 2 need not stabilize if* $k = (n-2)l$.

*Proof:* We show that it is possible that the system cycles forever through illegal states. See Figure 2 for an example for $n = 5$, $l = 2$, and $k = 6$. Let the initial state be as follows:

| state of machine 0 | state of machine 1 | state of machine 2 | state of machine 3 | state of machine 4 | privileged machines |
|---|---|---|---|---|---|
| 4 | 4 | 2 | 0 | 4 | 0,2,3,4 |
| 5 | 4 | 2 | 0 | 4 | 0,1,2,3,4 |
| 0 | 4 | 2 | 0 | 4 | 1,2,3,4 |
| 0 | 4 | 2 | 0 | 0 | 0,1,2,3 |
| 0 | 4 | 2 | 2 | 0 | 0,1,2,4 |
| 0 | 4 | 4 | 2 | 0 | 0,1,3,4 |
| **0** | **0** | 4 | 2 | 0 | 0,2,3,4 |
| **2** | **2** | **0** | 4 | 2 | 0,2,3,4 |
| **4** | **4** | **2** | **0** | 4 | 0,2,3,4 |
| **0** | **0** | **4** | **2** | **0** | 0,2,3,4 |

Only in the first round, all moves are shown. States indicated in bold would be colored blue according to the proof of Lemma 4.4.

Figure 2: Algorithm 2 using 6 states does not stabilize.

$S_0 = k-l = (n-3)l$, $S_1 = (n-3)l$, $S_2 = (n-4)l, \ldots, S_{n-2} = 0$, $S_{n-1} = k-l = (n-3)l$. Note that all machines except 1 are privileged. Let them move in the following order: $l$ moves by machine 0, one move by machine $n-1$, one by $n-2, \ldots$one by machine 1 (by that time it is privileged too). Call this sequence of moves a *round*. The resulting state is then 0, 0, $(n-3)l, \ldots, l$, 0, where again all machines except 1 are privileged. Note that for every $i$, the value of $S_i$ is increased by $l$. Now, do another round of moves. Again, all $S_i$ increase by $l$. Thus, after $n-2$ rounds, all $S_i$ are increased by $(n-2)l = k$. As the increase is done modulo $k$, all $S_i$ have their initial value again. Hence, the system can cycle forever in this illegal state and will never stabilize. □

**Theorem 4.8** *In the presence of a central demon it is necessary and sufficient to use* $(n-2)l+1$ *states for* $n \geq 3$ *for Algorithm 2.*

*Proof:* Follows from Corollary 4.6 and Lemma 4.7. □

In Figure 2 we show how the algorithm does not stabilize if it uses 6, i.e., $(n-2)l$ states, using the same initial configuration and the same order of moves as in Figure 1. (It is the same configuration if the initial state of $S_0$ and $S_4$ is defined as $k-l$ which is 4 in Figure 2 and 5 in Figure 1.) We indicated the states that are colored blue (see the proof of Lemma 4.4) in bold: the other (white) states are the "illegal states" that have to disappear from the system before it becomes stabilized (in case $k=7$). The difference between the two executions is as follows. In Figure 2 when the state of machine 4 becomes blue, this enables machine 0 as $0-0=0<2$, while in Figure 1 machine 0 is not enabled when the state of machine 4 becomes blue, as $(6-0)=6>2$. Although from some point in the execution of Figure 1 there are only 2 privileges, it is only the last state shown that is legal (because all state values lie within $l=2$ of each other).

It has been shown [14] that a minimum of three states are needed for a self-stabilizing mutual exclusion algorithm if $n>4$, and that two states suffice if $n\leq 4$. If one substitutes $l=1$ in our bound of $\max((n-2)l+1, l+1)$, one gets $k=n-1\geq 3$ for $n\geq 4$, and 2 for $n=2,3$.

**Bound for a Distributed Demon.** For the case of a distributed demon, the bound for $k$ is only one higher than in the case of a central demon, just as in Dijkstra's algorithm. We do not give the proofs which are all similar, but we just indicate where in the proof differences arise.

**Corollary 4.9** *In the presence of a distributed demon $(n-2)l+2$ states are sufficient for $n\geq 3$ for Algorithm 2.*

*Proof:* A difference with the proof of Lemma 4.4 occurs when the states are first colored blue. For the central demon, at that moment $S_0=S_1$ held, while for the distributed demon, machine 0 could have done a move simultaneous with machine 1, such that $S_0=S_1+1$ holds for the first blue states. Thus, the value of $S_0$ can be one higher when $S_{n-1}$ becomes blue too. Hence, in the case of a distributed demon $k\geq(n-2)l+2$ for $n\geq 3$ (and $k=2$ for $n=2$ and if $l\geq n$, one does not need any algorithm.) □

**Lemma 4.10** *Algorithm 2 need not stabilize in the presence of a distributed demon if $k=(n-2)l+1$.*

*Proof:* The proof is similar to that of Lemma 4.7. We illustrate this by an example for $n=5$ and $l=2$. Hence, we take $k=7$ and show a cycle of states which does not stabilize in Figure 3. □

**Theorem 4.11** *In the presence of a distributed demon it is necessary and sufficient to use $(n-2)l+2$ states for $n\geq 3$ for Algorithm 2.*

*Proof:* Follows from Corollary 4.9 and Lemma 4.10. □

| state of machine 0 | state of machine 1 | state of machine 2 | state of machine 3 | state of machine 4 | privileged machines |
|---|---|---|---|---|---|
| 5 | 4 | 2 | 0 | 5 | 0,1,2,3,4 |
| 6 | 4 | 2 | 0 | 5 | 0,1,2,3,4 |
| **0** | **6** | 4 | 2 | 0 | 0,1,2,3,4 |
| **1** | **6** | 4 | 2 | 0 | 0,1,2,3,4 |
| **2** | **1** | **6** | 4 | 2 | 0,1,2,3,4 |
| **3** | **1** | **6** | 4 | 2 | 0,1,2,3,4 |
| **4** | **3** | **1** | **6** | 4 | 0,1,2,3,4 |
| **5** | **3** | **1** | **6** | 4 | 0,1,2,3,4 |
| **6** | **5** | **3** | **1** | **6** | 0,1,2,3,4 |
| **0** | **5** | **3** | **1** | **6** | 0,1,2,3,4 |
| **1** | **0** | **5** | **3** | **1** | 0,1,2,3,4 |
| **2** | **0** | **5** | **3** | **1** | 0,1,2,3,4 |
| **3** | **2** | **0** | **5** | **3** | 0,1,2,3,4 |
| **4** | **2** | **0** | **5** | **3** | 0,1,2,3,4 |
| **5** | **4** | **2** | **0** | **5** | 0,1,2,3,4 |

Alternately, machine 0 and all machines move. States indicated in bold would be colored blue according to the proof of Corollary 4.9.

Figure 3: Algorithm 2 using 7 states does not stabilize for a distributed demon.

# 5 Extended Algorithm

One drawback of the previous algorithms is that privileges can get held up at one process if a privilege is used in an application for something time-consuming, such as doing some work inside a critical section. If one process, $i$, has a privilege and enters the critical section, the process will not pass the privilege for a length of time. During this time, other privileges may propagate around the system and get held up at $i$. For example, if $l = 5$, and the system looks like 555555500000000, then machine seven has the only privilege. Even though only one process is in its critical section, no other process can enter its critical section until this privilege is passed to the right. Intuitively, one could say that in the case above, machine seven holds all five privileges, while it needs only one. Thus, we introduce the concept of the *number of privileges* that a machine holds. For clarity's sake, we temporarily return to unbounded integers.

For machine $i \neq 0$ the number of privileges is $S_{i-1} - S_i$ if $S_{i-1} \geq S_i$ and 1 otherwise, for machine 0 the number of privileges is $l - (S_0 - S_{n-1})$ if $|S_0 - S_{n-1}| < l$ and 0 otherwise. For machine 0 the number of privileges thus equals the number of moves it can make in Algorithm 1 while $S_{n-1}$ is not changed. The solution to the problem stated above is straightforward: a machine can choose to pass all its privileges to the right or, if it wants to enter its critical section, first pass any excess privileges and keep only one. Clearly, a machine should also be able to pass on any excess privileges if they arise while it is in its critical section. Using the definition of the number of privileges held by one machine, it is

easy to deduce what the algorithm looks like for the unbounded case.

We do not give any details about entering or returning from the critical section, for the exclusion algorithm per se it is sufficient to know that the process has a choice. As we do not make any assumptions about this choice, we just let the demon make the choice, as it has no effect on the algorithm itself. In an actual implementation, the choice should be the processes' choice, of course.

For machine 0 passing all its privileges to the right means increasing $S_0$ by 1 so many times that it is privileged no more, i.e., increasing $S_0$ to $S_{n-1} + l$. If machine 0 has more than one privilege ($-l < S_0 - S_{n-1} < l - 1$), it can choose to keep one and pass the rest, i.e., increase $S_0$ to $S_{n-1} + l - 1$. For other machines $i$, passing all privileges remains the same as in Algorithm 1, while if it has more than one privilege ($|S_i - S_{i-1}| > 1$) keeping one means setting $S_i$ to $S_{i-1} - 1$. Thus, Algorithm 1' becomes:

{Algorithm 1': Unbounded $l$-exclusion algorithm without privilege hold-ups}
{for machine 0}
**begin**
$(|S_0 - S_{n-1}| < l) \longrightarrow S_0 := S_{n-1} + l$
$(-l < (S_0 - S_{n-1}) < l - 1) \longrightarrow S_0 := S_{n-1} + l - 1$
**end**
{for other machines, $i$}
**begin**
$(S_i \neq S_{i-1}) \longrightarrow S_i := S_{i-1}$
$(|S_i - S_{i-1}| > 1) \longrightarrow S_i := S_{i-1} - 1$
**end**

Reverting to bounded numbers, we first define the number of privileges of one process:
for machine 0 the number of privileges is
$l - ((S_0 - S_{n-1}) \ mod \ k)$ if $(S_0 - S_{n-1}) \ mod \ k < l$ and 0 otherwise,
for machine $i \neq 0$ the number of privileges is $(S_{i-1} - S_i) \ mod \ k$.
The changes in the algorithm are now straightforward:

{Algorithm 3: Extended Algorithm}
{for machine 0}
**begin**
$((S_0 - S_{n-1}) \ mod \ k < l) \longrightarrow S_0 := (S_{n-1} + l) \ mod \ k$
$((S_0 - S_{n-1}) \ mod \ k < l - 1) \longrightarrow S_0 := (S_{n-1} + l - 1) \ mod \ k$
**end**
{for other machines, $i$}
**begin**
$(S_i \neq S_{i-1}) \longrightarrow S_i := S_{i-1}$
$((S_{i-1} - S_i) \ mod \ k > 1) \longrightarrow S_i := (S_{i-1} - 1) \ mod \ k$
**end**

The sets $G_3$ of all possible global states and $L_3$ of all legal states are the same as for Algorithm 2, thus $G_3 = G_2$ and $L_3 = L_2$ (see Section 4).

## 5.1 Correctness Proof

Note that Algorithm 3 only differs from Algorithm 2 if $l > 1$. Thus we assume in the sequel that $l \geq 2$. The correctness proof of Algorithm 3 is similar again, however, the bounds for the number $k$ increase. We summarize the first three Lemmas into one:

**Lemma 5.1** *Given an arbitrary global state, eventually every machine will make a move.*

*Proof:* Again, it is not possible that no machine holds a privilege, and each machine can do at most two moves before another move of its left neighbor. (If $k > l$). □

**Lemma 5.2** *After a finite number of moves, the system will be in a legal state.*

*Proof:* Similar to the proof of Lemma 4.4. See Figure 4 for an example for $n = 5$, $l = 2$, and $k = 13$. Consider the first move by machine 1 after a move by machine 0. Color

| state of machine 0 | state of machine 1 | state of machine 2 | state of machine 3 | state of machine 4 | moving machines |
|---|---|---|---|---|---|
| 2 | 11 | 7 | 4 | 2 | 0,4,3,2,1 |
| **4** | **3** | 11 | 7 | 4 | 0,4 |
| **6** | **3** | 11 | 7 | 6 | 0,4,3 |
| **8** | **3** | 11 | 10 | 7 | 0,4,3,2,1 |
| **9** | **8** | **2** | 11 | 9 | 0,4,3 |
| **11** | **8** | **2** | **1** | 11 | 0,4,3,2,1 |
| **0** | **12** | **7** | **2** | **0** | 0,4,3 |
| **2** | **12** | **7** | **6** | **2** | 0,4,3,2,1 |
| **4** | **3** | **11** | **7** | **5** | 4,3 |
| **4** | **3** | **11** | **10** | **7** | 4,3,2 |
| **4** | **3** | **2** | **11** | **9** | 4,3,2,1 |
| **4** | **4** | **3** | **1** | **11** | 4,3 |
| **4** | **4** | **3** | **3** | **0** | 4,2 |
| **4** | **4** | **4** | **3** | **3** | 0,3 |

Not all moves are shown separately. States indicated in bold are colored blue according to the proof of Lemma 5.2. This particular execution stabilizes for a central demon.

Figure 4: Algorithm 3 using 13 states.

the resulting states of machines 0 and 1 blue, and the other states white. Without loss of generality, assume that the value of $S_0$ is $n-1$. Then, $S_1$ is either $n-1$ or $n-2$. Afterwards, color all states as in Lemma 4.4. Let $h$ be the value of $S_0$ when $S_{n-1}$ eventually becomes blue. The question is, how large can $h$ become in the worst case. When the states are first colored, there are at most $n-2$ different white values present. For the value originally in

$S_{n-1}$, that can cause an increase in $S_0$ by at most $l$. However, the value originally present in $S_{n-2}$ can give rise to two consecutive values in $S_{n-1}$, and hence, an increase of at most $l+1$ in $S_0$. In general, a white value (say, $v$) originally in machine $j$ can give rise to $n-j$ consecutive (white) values in $S_{n-1}$ (namely, $v-(n-1-j), \ldots, v$). This can cause an increase of at most $n-j+l-1$ in $S_0$. Summing over machines 2 to $n-1$, this gives an increase of $\sum_{j=2}^{n-1}(n-j+l-1) = (n-2)(n-3+2l)/2 = (n-2)l+(n-2)(n-3)/2$. As the initial blue value of $S_0$ was $n-1$, we have that $h \leq n-1+(n-2)l+(n-2)(n-3)/2$ when $S_{n-1}$ becomes blue for the first time. This blue value can be as low as 0 (a decrease by 1 from the original $n-1$ for each move to the right). To avoid a possible cycle of illegal states, we demand that $k > h$, or $k \geq n+(n-2)l+(n-2)(n-3)/2$. As in Lemmas 3.4 and 4.4, the blue states form a non-increasing sequence. When eventually machine 0 moves again, there are only $l$ privileges left (see Lemma 4.4). □

**Theorem 5.3** *Algorithm 3 stabilizes in the presence of a central demon.*

*Proof:* Similar to the proof of Theorem 4.5. □

## 5.2 Value of the Bound

**Corollary 5.4** *In the presence of a central demon $n+(n-2)l+(n-2)(n-3)/2$ states are sufficient for $n \geq 3$ for Algorithm 3.*

*Proof:* The bounds derived for $k$ during the proof are $k \geq l+1$ (Lemma 5.1) and $k > n-1+(n-2)l+(n-2)(n-3)/2$ (Lemma 5.2). Hence, the proof above is valid for $k \geq n+(n-2)l+(n-2)(n-3)/2$ if $n > 2$. Note that for $n = 2$ no algorithm is needed, as $l \geq 2$. □

**Lower bounds.** Unfortunately, we do not know whether the bound derived above is exact. We expect it is not, as the execution sketched in the proof of Lemma 5.2 suggests that the system might cycle for a value of $k = n-1+(n-2)l+(n-2)(n-3)/2$. However, if we take $n = 5$, $l = 2$, and $k = 13$, this execution does stabilize later on (see Figure 4).

We develop a series of strategies to arrive at a value of $k$ which can cause Algorithm 3 to cycle indefinitely in illegal states. It depends on the relative value of $n$ and $l$ which of these strategies yields the largest value of $k$. Summarizing, we then give one expression for a lower bound for $k$.

**Lemma 5.5** *Algorithm 3 need not stabilize if $k = n-1+(n-2)l$.*

*Proof:* We show that it is possible that the system cycles forever through illegal states. See Figure 5 for an example for $n = 5$, $l = 2$, and $k = 10$. Let the initial state be as follows: $S_0 = 0$, $S_1 = (n-2)(l+1)$, $S_2 = (n-3)(l+1)$, $\ldots$, $S_{n-2} = l+1$, $S_{n-1} = 0$. Define a round of moves as follows: one move of machine 0 where it passes all privileges (i.e., an increase by $l$ of $S_0$), followed by one move by machines $n-1, n-2, \ldots, 1$, where they keep one privilege and pass the rest. As a consequence, all state values are increased by $l$ after one round of moves. (For machine 1, this is the case because $(n-2)(l+1)+l = n-1+(n-2)l+l-1 = l-1 \ mod\ k$). Note that all machines hold privileges. Hence, each round of moves as defined above add $l \pmod k$ to each state and the system will cycle forever. □

However, we can do better than that if $n$ is sufficiently large compared to $l$.

| state of machine 0 | state of machine 1 | state of machine 2 | state of machine 3 | state of machine 4 | moving machines |
|---|---|---|---|---|---|
| 0 | 9 | 6 | 3 | 0 | 0 |
| **2** | 9 | 6 | 3 | 0 | 4 |
| **2** | 9 | 6 | 3 | 2 | 3 |
| **2** | 9 | 6 | 5 | 2 | 2 |
| **2** | 9 | 8 | 5 | 2 | 1 |
| **2** | **1** | 8 | 5 | 2 | 0,4,3,2,1 |
| 4 | 3 | **0** | 7 | 4 | 0,4,3,2,1 |
| 6 | 5 | 2 | **9** | 6 | 0,4,3,2,1 |
| 8 | 7 | 4 | 1 | **8** | 0,4,3,2,1 |
| 0 | 9 | 6 | 3 | 0 | 0,4,3,2,1 |

Only in the first round, all moves are shown. Values generated by the same value (2) of machine 0 are indicated in bold.

Figure 5: Algorithm 3 using 10 states does not stabilize.

**Lemma 5.6** *Algorithm 3 need not stabilize if* $k = n-1+(n-2)l+(n-2-l)l$ *(if* $l \leq n-2$*).*

*Proof:* We show that it is possible that the system cycles forever through illegal states. See Figure 6 for an example for $n = 5$, $l = 2$, and $k = 12$. Let the initial state be as follows: $S_0 = 0$, $S_1 = (n-2-l)(2l+1) + l^2 + l$, $S_2 = (n-3-l)(2l+1) + l^2 + l$, ..., $S_{n-1-l} = l^2 + l$, ($S_{n-1-l+1} = l^2 + l - 2l$, $S_{n-1-l+2} = l^2 + l - 2l - 2$, ...,) $S_{n-1} = 0$. For even values of $l$, $S_{n-1} = 0$ because $l^2 + l - (2l+2)l/2 = l^2 + l - (l+1)l = 0$, while for odd values of $l$, $S_{n-1} = 0$ because $l^2 + l - 2l(l+1)/2 = l^2 + l - l^2 - l = 0$. We now have two different rounds of moves: even and odd rounds. An even round consists of a move for all machines, where machine 0 passes all privileges, machines 1, ..., $n-1-l$ keep one privilege, while the last $l$ machines keep one or pass all privileges alternately. An odd round consists of a move by machine 0 (all privileges passed), and moves by the last $l$ machines, now pass all or keep one privilege alternately. Let us call an odd round followed by an even round a *combined round* (*c-round* for short). The effect of a c-round is that all state values are increased by $2l$ mod $k$. (For machine 1, this is the case because $(n-2-l)(2l+1) + l^2 + l + 2l = n - 1 + (n-2)l + (n-2-l)l + 2l - 1 = 2l - 1$ *mod* $k$ (machine 0 has moved twice already)). What happens exactly is best illustrated if we follow one state value as it is passed from machine 0 all the way to machine $n-1$. (See the bold numbers in Figure 6.) Let machine 0 have value $m$ when machine 1 makes a move. As a result, $S_1 = m - 1$. As long as this value resides in a machine with a number less than $n-1-l$, it is moved to the right and decreased once in a c-round. In machine $n-1-l$ the value has become $m - (n-1-l)$. From then on, the value is "split" as it were, in a sequence of values which are further decreased, and a sequence where the value remains equal. When the value of the first sequence arrives at machine $n-1$, the value has become $m - (n-1-l) - l$, while in the next round, the value of the second sequence arrives unaltered by the last $l$ machines as $m - (n-1-l)$. Hence, machine 0 can move

| state of machine 0 | state of machine 1 | state of machine 2 | state of machine 3 | state of machine 4 | moving machines |
|---|---|---|---|---|---|
| **0** | **11** | 6 | 2 | 0 | 0,4,3 |
| 2 | **11** | 6 | 5 | 2 | 0,4,3,2,1 |
| 4 | 3 | **10** | 6 | 4 | 0,4,3 |
| 6 | 3 | **10** | **9** | 6 | 0,4,3,2,1 |
| 8 | 7 | 2 | **10** | **8** | 0,4,3 |
| 10 | 7 | 2 | 1 | **10** | 0,4,3,2,1 |
| 0 | 11 | 6 | 2 | 0 | 0,4,3 |

Alternately, machines 0, 4, 3, and all machines move. Values generated by the same value (0) of machine 0 are indicated in bold.

Figure 6: Algorithm 3 using 12 states does not stabilize.

and increase its value with $l$ twice also. Hence, each c-round of moves as defined above adds $2l$ $(mod\ k)$ to each state and the system will cycle forever. □

In the first approach (Lemma 5.5) each value that occurs in $S_1$ generates one value in $S_{n-1}$. In the second approach (Lemma 5.6) each value in $S_1$ generates two values in $S_{n-1}$ which are $l$ apart. It is possible to let one value in $S_1$ generate $m$ values in $S_{n-1}$, each $l$ apart, if $(m-1)l \leq n-2$. We first show how to do this for $m = 3$ before handling the general case.

**Lemma 5.7** *Algorithm 3 need not stabilize if* $k = n-1+(n-2)l+(n-1-2l)2l-l$ *(if* $2l \leq n-2$*).*

*Proof:* We show that it is possible that the system cycles forever through illegal states. See Figure 7 for an example for $n = 7$, $l = 2$, and $k = 22$. Let the initial state be as follows: $S_0 = 0$, $S_1 = (n-2-2l)(3l+1)+2l^2+2l$, $S_2 = (n-3-2l)(3l+1)+2l^2+2l$, $\ldots$, $S_{n-1-2l} = 2l^2+2l$, ($S_{n-1-2l+1} = 2l^2+2l-3l$, $S_{n-1-2l+2} = 2l^2+2l-3l$, $S_{n-1-2l+3} = 2l^2+2l-3l-3, \ldots$), ($S_{n-1-2l+3i+1} = 2l^2+2l-3(i+1)l$, $S_{n-1-2l+3i+2} = 2l^2+2l-3(i+1)l-l$, $S_{n-1-2l+3i+3} = 2l^2+2l-3(i+1)l-2l+(2l-3i-3), \ldots$), $S_{n-1} = 0$. We now have three different rounds of moves: 0-rounds, 1-rounds, and 2-rounds. Machines 0 and $n-l$, $\ldots$, $n-1$ move in all rounds (the latter machines keeping one privilege in every third move), and machines 1, $\ldots$, $n-1-2l$ move only in a 1-round, keeping one privilege. Machines $n-2l$, $\ldots$, $n-2-l$ alternately move while keeping one privilege, move and pass all privileges and do not move. For a machine $n-1-2l+i$ $(1 \leq i \leq l-1)$ no move is made in the $(i+1\ mod\ 3)$-round. Finally, machine $n-1-l$ does not move in the same round as machine $n-2-l$ does not, passes all privileges in the next round and keeps one in the round after that. Let us call a 0-round followed by a 1-round followed by a 2-round a *combined round* (*c-round* for short). The effect of a c-round is that all state values are increased by $3l$ $mod\ k$. (For machine 1, this is the case because $(n-2-2l)(3l+1)+2l^2+2l+3l = n-1+(n-2)l+(n-1-2l)2l-l+2l-1 = 2l-1\ mod\ k$

| state of mch. 0 | state of mch. 1 | state of mch. 2 | state of mch. 3 | state of mch. 4 | state of mch. 5 | state of mch. 6 | moving machines |
|---|---|---|---|---|---|---|---|
| 0 | 19 | 12 | 6 | 4 | 3 | 0 | 0,6,5,4,3 |
| 2 | 19 | 12 | 11 | 6 | 4 | 2 | 0,6,5,4,3,2,1 |
| **4** | **3** | 18 | 12 | 10 | 6 | 4 | 0,6,5 |
| 6 | **3** | 18 | 12 | 10 | 9 | 6 | 0,6,5,4,3 |
| 8 | **3** | 18 | 17 | 12 | 10 | 8 | 0,6,5,4,3,2,1 |
| 10 | 9 | **2** | 18 | 16 | 12 | 10 | 0,6,5 |
| 12 | 9 | **2** | 18 | 16 | 15 | 12 | 0,6,5,4,3 |
| 14 | 9 | **2** | **1** | 18 | 16 | 14 | 0,6,5,4,3,2,1 |
| 16 | 15 | 8 | **2** | **0** | 18 | 16 | 0,6,5 |
| 18 | 15 | 8 | **2** | **0** | **21** | 18 | 0,6,5,4,3 |
| 20 | 15 | 8 | 7 | **2** | **0** | **20** | 0,6,5,4,3,2,1 |
| 0 | 21 | 14 | 8 | 6 | **2** | **0** | 0,6,5 |
| 2 | 21 | 14 | 8 | 6 | 5 | **2** | 0,6,5,4,3 |

Alternately, machines 0, 6, 5; machines 0, 6, 5, 4, 3; and all machines move. Values generated by the same value (4) of machine 0 are indicated in bold. (Only about one third of the complete cycle is shown.)

Figure 7: Algorithm 3 using 22 states does not stabilize.

(machine 0 has moved twice already)). What happens exactly is best illustrated if we follow one state value as it is passed from machine 0 all the way to machine $n-1$. (See the bold numbers in Figure 7.) Let machine 0 have value $x$ when machine 1 makes a move. As a result, $S_1 = x - 1$. As long as this value resides in a machine with a number less than $n-1-2l$, it is moved to the right and decreased once in a c-round. In machine $n-1-2l$ the value has become $x-(n-1-2l)$. From then on, the value is "split" as it were, in a sequence of values which are further decreased, and two sequences of values which remain the same until machine $n-1-l$. In this machine, the value in the first sequence has become $x-(n-1-l)$. In the last $l$ machines this is decreased further to $x-(n-1)$ in the first sequence, while in the second sequence it remains unaltered at $x-(n-1-l)$. In the third sequence, the value of $x-(n-1-2l)$ is handed down, such that the three values which appear in machine $n-1$ are $l$ apart and machine 0 can increase its value by $l$ three times. □

**Theorem 5.8** *Let $m$ be the rounded result of $(n-1)/(2l)$. Algorithm 3 need not stabilize if $k = n-1+(n-2)l+(n-1-ml)ml-l$ (if $m \geq 1$).*

*Proof:* The generalization of Lemmas 5.6 and 5.7 is that if $n$ is sufficiently large compared to $l$ (i.e., $ml \leq n-2$) it is possible to construct a cycle of illegal states where one value in machine 0 is split in $m+1$ sequences of values which generate $m+1$ values in machine $n-1$ that are $l$ apart. In a combined round of $m+1$ rounds, all state values are increased

by $(m+1)l$. This can be achieved for a value of $k = ml(n-1-ml)+n-1+(n-2)l-l$. (In the first $n-1-ml$ machines the initial values are $(m+1)l+1$ apart, while a value of $m(l^2+l)$ in machine $n-1-ml$ generates the values $0, l, \ldots, ml$ in machine $n-1$.) To obtain the value of $m$ which yields the largest value of $k$, consider $k$ as a function of $m$. For this quadratic function the maximum is attained for a value of $m$ such that $l(n-1)-2ml^2 = 0$, or $m = (n-1)/(2l)$. As $m$ has to be an integer, we have to round $(n-1)/(2l)$. □

**Corollary 5.9** *For values of $n$ and $l$ such that $n-1 \geq 2l$ and $n-1$ divisible by $2l$, Algorithm 3 need not stabilize for $k = (n-1)^2/4+(n-3)l+n-1$.*

**Bounds for a Distributed Demon.** For the case of a distributed demon, the bounds for $k$ are $l$ higher than in the case of a central demon. We do not give the proofs which are all similar, but we just indicate where in the proof differences arise.

**Corollary 5.10** *In the presence of a distributed demon $n+(n-1)l+(n-2)(n-3)/2$ states are sufficient for Algorithm 3.*

*Proof:* The only difference with the proof of Lemma 5.2 occurs when the states are first colored blue. For the central demon, at that moment $S_0 = S_1$ or $S_0 = S_1 - 1$ held, while for the distributed demon, machine 0 could have done a move simultaneous with machine 1, such that $S_0 = S_1 + l$ or $S_0 = S_1 - 1 + l$ holds for the first blue states. Thus, the value of $S_0$ can be $l$ higher when $S_{n-1}$ becomes blue too. Hence, in the case of a distributed demon $k \geq n+(n-1)l+(n-2)(n-3)/2$ (for $n \geq 3$). □

**Theorem 5.11** *Let $m$ be the rounded result of $(n-1)/(2l)$. Algorithm 3 need not stabilize in the presence of a distributed demon if $k = n-1+(n-2)l+(n-1-ml)ml$ (if $m \geq 1$).*

*Proof:* The proof is similar to that of Theorem 5.8. However, as in the case of Corollary 5.10, the initial value of $S_0$ can be $l$ higher, causing an increase of $l$ for the value of $k$. □

**Corollary 5.12** *For values of $n$ and $l$ such that $n-1 \geq 2l$ and $n-1$ divisible by $2l$, Algorithm 3 need not stabilize in the presence of a distributed demon for $k = (n-1)^2/4+(n-2)l+n-1$.*

## 5.3 Comparison Between Algorithm 2 and Algorithm 3

Algorithm 3 corrects the problem of holding up privileges. In Algorithm 2, if one process does not pass a privilege for a while, other privileges can also be stopped at this point. If one process is particularly slow, this process can be a bottleneck because many privileges are held up at this process. If all processes take similar times to execute the critical section, this problem will not be a serious drawback. Algorithm 3 combats this problem by adding the possibility to pass excess privileges before or while a process is in its critical section. Thus, excess privileges are never held up and the bottleneck is avoided.

The number of states required by the algorithms is also different. Whereas for Algorithm 2 the number of states ($k$) required is $O(nl)$, for Algorithm 3 we need $O(n^2)$. Stabilization time is difficult to compare for the two algorithms, it depends on the time

needed inside the critical section relative to the time needed for a move. If we just count the number of moves, stabilization time of Algorithm 3 is worse. However, in Algorithm 2 privileges can be held up, and moves have to wait until a process has returned from its critical section.

# 6 Conclusions

This paper presents self-stabilizing l-exclusion algorithms. Many early papers dealing with self-stabilization [2, 4, 5, 8] solved the mutual exclusion problem. This paper solves a more general problem (the l-exclusion problem) in a self-stabilizing way. It is more general because the mutual exclusion problem is just a specific case of the l-exclusion problem ($l = 1$). The algorithms presented in this paper only require a process to be able to examine the state of its left neighbor. Once the system stabilizes, there will be at most $l$ privileges in the system (there may be less).

There are many advantages of having a self-stabilizing algorithm instead of an algorithm that is not self-stabilizing. The initial global state of the system can be arbitrary. This means that no initialization of the states of the processes needs to be done. Once the system is started, in a finite amount of time, the system will be in a legal state (no more than $l$ privileges will exist). Another advantage is the inherent fault-tolerance of the system. If a fault occurs which causes the system to be put in an illegal state, the system will again converge to a legal state. These faults can be machine failures where a machine halts and restarts in an unknown state, or it could be a communication failure where some messages are lost or corrupted. These messages may cause processes to receive an incorrect state value. The system will correct itself and converge to a legal state.

The algorithms presented in this paper are self-stabilizing solutions to the $l$-exclusion problem. They are fault-tolerant because of the self-stabilizing property. They are also easier to implement because they only require that a process be able to examine the state of its left neighbor. The implementation can be done through either message passing or shared variables. So, the algorithms are versatile and useful.

# References


[1] Afek Y., Dolev D., Gafni E., Merritt M., and Shavit N. "A Bounded First-In, First-Enabled Solution to the l-Exclusion Problem," *4th International Workshop on distributed Algorithms*, Bari, Italy, September 24-26, 1990, Lecture Notes in Computer Science, Vol. 486, pp. 422-431.

[2] Brown G., Gouda M., and Wu M. "Token Systems that Self Stabilize," *IEEE Trans. on Comput.*, Vol. 38, No. 6, June 1989, pp. 845-852.

[3] Burns J., Gouda M., and Miller R. "On Relaxing Interleaving Assumptions," *MCC Workshop on Self-Stabilization*, Austin, Texas, November 1989.

[4] Burns J. and Pachl J. "Uniform Self-Stabilizing Rings," *ACM Trans. Prog. Lang. and Syst.*, Vol. 11, No. 2, 1989, pp. 330-344.

[5] Dijkstra E.W. "Self-Stabilization in Spite of Distributed Control," *Communications of the ACM 17*, 1974, pp. 643-644.

[6] Dijkstra E.W. "Self-Stabilization in Spite of Distributed Control," in *Selected writings on computing: a personal perspective*, Springer-Verlag, Berlin, 1982, pp. 41-46.

[7] Dolev D., Gafni E., and Shavit N. "Toward a Non-Atomic Era : $l$-Exclusion as a Test Case," *20th Annual ACM Symposium on Theory of Computing*, Chicago, IL, May 2-4, 1988, pp. 78-92.

[8] Dolev S., Israeli A., and Moran S. "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity," *9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Quebec, August 22-24, 1990, pp. 103-118.

[9] Fischer M., Lynch N., Burns J., and Borodin A. "Resource Allocation with Immunity to Limited Process Failure," *20th IEEE Symp. on Foundations of Computer Science*, pp. 234-254, 1979.

[10] Fischer M., Lynch N., Burns J., and Borodin A. "Distributed Fifo Allocation of Identical Resources Using Small Shared Space," *ACM Transactions on Programming Languages and Systems*, 11(1): 90-114, January 1989.

[11] Katz, S., and Perry K. J. "Self-Stabilizing Extensions for message-passing Systems," *9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Quebec, August 1990, pp. 91-101.

[12] Lin, C., and Simon J. "Observing Self-Stabilization," *11th Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992, pp. 113-123.

[13] Peterson, G. "Observations on l-exclusion," *28th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL, October 3-5, 1990, pp. 568-577.

[14] Tchuente M. "Sur l'auto-stabilisation dans un reseau d'ordinateurs," *RAIRO Informatique Theorique 15*, No. 1, pp.47-66, 1981.