17.3 STABILIZING MUTUAL EXCLUSION

Dijkstra's work [D74] initiated the field of self-stabilization in distributed systems. He first demonstrated its feasibility by proposing stabilizing solutions to the problem of mutual exclusion on three different types of networks. In this section, we present the solutions to two of these versions.

17.3.1 Mutual Exclusion on a Unidirectional Ring

Consider a *unidirectional ring* of n processes 0, 1, 2, …, $n-1$ (Figure 17.2). Each process can remain in one of the k possible states 0, 1, 2, …, $k-1$. We consider the shared-memory model of computation: A process i, in addition to reading its own state $s[i]$, can read the state $s[i-1 \bmod n]$ of its *predecessor* process $i-1 \bmod n$. Depending on whether a predefined guard (which is a Boolean function of these two states) is true, process i may choose to modify its own state.

We will call a process with an enabled guard a *privileged process* or a process *holding a token*. This is because a privileged process is one that can take an action, just as in a token ring network, a process holding the token is eligible to transmit or receive data. A legal configuration of the ring is characterized by the following two conditions:

Safety: The number of processes with an enabled guard is exactly one.

Liveness: During an infinite behavior, the guard of each process is enabled infinitely often.

A privileged process executes its critical section. A process that has an enabled guard but does not want to execute its critical section simply executes an action to pass the privilege to its neighbor. Transient failures may transform the system to an illegal configuration. The problem is to design a protocol, so that starting from an arbitrary initial state, the system eventually converges to a legal configuration and remains in that configuration thereafter.

Dijkstra's solution assumed process 0 to be a *distinguished process* that behaves differently from the remaining processes in the ring. All the other processes run identical programs. There is a central scheduler for the entire system. In addition, the condition $k > n$ holds. The programs are as follows:

```
Program ring;
{program for process 0}
do s[0]=s[n−1]→s[0]:=s[0]+1 mod k   od
{program for process i≠0}
do s[i]≠s[i−1]→s[i]:=s[i−1] od
```

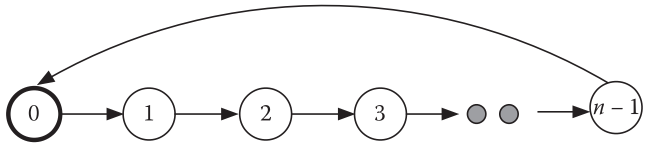

FIGURE 17.2 A unidirectional ring of n processes.

Before studying the proof of correctness of the aforementioned algorithm, we *strongly* urge the reader to study a few sample runs of the aforementioned protocol and observe the convergence and closure properties. A configuration in which $\forall i, j : s[i] = s[j]$ is an example of a legal configuration.

Proof of correctness

Lemma 17.1

[No deadlock] In any configuration, at least one process must have an enabled guard.

Proof: If every process 1 through $n - 1$ has a disabled guard, then $\forall i > 0, s[i] = s[i - 1]$. But this implies that $s[0] = s[n - 1]$, so process 0 must have an enabled guard. ■

Lemma 17.2

[Closure] The legal configuration satisfies the closure property.

Proof: If only process 0 has an enabled guard, then $\forall i, j : 0 \le i, j \le n - 1 : s[i] = s[j]$. A move by process 0 will disable its own guard and enable the guard for process 1. If only process i $(0 < i < n - 1)$ has an enabled guard, then

- $\forall j < i : s[j] = s[i - 1]$
- $\forall k > i : s[k] = s[i]$
- $s[i] \ne s[i - 1]$

Accordingly, a move by process i will disable its own guard and enable the guard for process $(i + 1)$ mod n. Similar arguments hold when only process $(n - 1)$ has an enabled guard. ■

As a consequence of Lemmas 17.1 and 17.2, in an infinite computation, the guard of each process will be true infinitely often.

Lemma 17.3

[Convergence] Starting from any illegal configuration, the ring eventually converges to a legal configuration.

Proof: Observe that every action by a process disables its own guard and enables *at most* one new guard in a different process—so the number of enabled guards never increases. Now, assume that the claim is false, and the number of enabled guards remains constant during an infinite suffix of a behavior. This is possible if every action that disables an existing guard enables exactly one new guard.

There are n processes with $k(k > n)$ states per process. By the pigeonhole principle, in any initial configuration, at least one element $j \in \{0, 1, 2, \ldots, k - 1\}$ *must not* be the initial state of any process. Each action by process ($i > 0$) essentially copies the state of its predecessor, so if j is not the state of any process in the initial configuration, no process can be in state j until $s[0]$ becomes equal to j. However, it is guaranteed that at some point, $s[0]$ will be equal to j, since process 0 executes actions infinitely often, and every action increments $s[0]$ (mod k). Once $s[0] = j$, eventually every process attains the state j, and the system reaches a legal configuration. ■

The property of stabilization follows from Lemmas 17.2 and 17.3.

17.3.2 Mutual Exclusion on a Bidirectional Array

The second protocol operates on an array of processes 0 through $n - 1$ (Figure 17.3). We present here a modified version of Dijkstra's protocol, taken from [G93]

In this system, $\forall i : s[i] \in \{0, 1, 2, 3\}$ and is independent of the size of the array. The two processes 0 and $n - 1$ behave differently from the rest—they have two states each. By definition, $s[0] \in \{1, 3\}$ and $s[n-1] \in \{0, 2\}$. Let $N(i)$ designate the set of neighbors of process i. The program is as follows:

```
program four-state;
{program for process i, i = 0 or n - 1}
do ∃j ∈N(i):s[j]=s[i]+1mod4→s[i]:=s[i]+2mod4 od
{program for process i, 0 < i < n - 1}
do ∃j ∈N(i):s[j]=s[i]+1mod4→s[i]:=s[i]+1mod4 od
```

Proof of correctness

The *absence of deadlock* can be trivially demonstrated using arguments similar to those used in Lemma 17.1. We focus on convergence only.

For a process i, call the processes $i + 1$ and $i - 1$ to be the *right* and the *left* neighbors, respectively. Define two predicates $L \cdot i$ and $R \cdot i$ as follows:

$$L \cdot i \equiv s[i-1] := s[i] + 1 \bmod 4$$

$$R \cdot i \equiv s[i+1] := s[i] + 1 \bmod 4$$

We will represent $L \cdot i$ by drawing a → from process $i - 1$ to process i and $R \cdot i$ by drawing a ← from process $i + 1$ to process i. Thus, any process with an enabled guard has at least one arrow pointing toward it.

For a process i ($0 < i < n - 1$), the possible moves fall exactly into one of the seven cases in part (a) of Table 17.1. Each entry in the columns *precondition* and *postcondition* represents

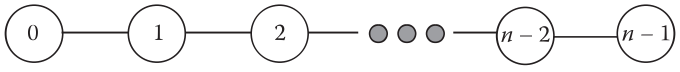

FIGURE 17.3 An array of n processes.

TABLE 17.1 Possible Changes Caused due to an Action by Process i

Case	**Precondition**	**Postcondition**
(a) $0 < i < n - 1$		
a	$x + 1 \rightarrow x, x$	$x + 1, x + 1 \rightarrow x$
b	$x + 1 \rightarrow x \leftarrow x + 1$	$x + 1, x + 1, x + 1$
c	$x + 1 \rightarrow x, x + 2$	$x + 1, x + 1 \leftarrow x + 2$
d	$x + 1 \rightarrow x \rightarrow x + 3$	$x + 1, x + 1, x + 3$
e	$x,x \leftarrow x + 1$	$x \leftarrow x + 1, x + 1$
f	$x + 2, x \leftarrow x + 1$	$x + 2 \rightarrow x + 1, x + 1$
g	$x + 3 \leftarrow x \leftarrow x + 1$	$x + 3, x + 1, x + 1$
Case	**Precondition**	**Postcondition**
(b) $i = 0$ and $i = n - 1$		
h	$x \leftarrow x + 1$	$x + 2 \rightarrow x + 1$
k	$x + 1 \rightarrow x$	$x + 1 \leftarrow x + 2$

the states of the three processes ($i - 1$, i, $i + 1$) before and after the action by process i. Note that $x \in \{0, 1, 2, 3\}$, and all + operations are mod 4 operations.

Case (a) represents the move when a $\rightarrow$ is transferred to the right neighbor. Case (e) shows the move when a $\leftarrow$ is transferred to the left neighbor. Cases (c) and (f) show how a $\rightarrow$ can be converted to a $\leftarrow$, and vice versa. Finally, cases (b), (d), and (g) correspond to moves by which the number of enabled guards is reduced. Note that in all seven cases, the total number of enabled guards is always nonincreasing.

Part (b) of Table 17.1 shows a similar list for the two processes 0 and $n - 1$. Case (h) lists the states of processes 0 and 1, and case (k) lists the states of processes $n - 2$ and $n - 1$. Process 0 transforms a $\leftarrow$ to a $\rightarrow$, and process $n - 1$ transforms a $\rightarrow$ to a $\leftarrow$. These two processes thus act as *reflectors*. Once again, the number of enabled guards does not increase.

Lemma 17.4

If the number of enabled guards in the processes $0..i$ ($i < n - 1$) is positive, and process $i + 1$ does not execute an action, then after at most *three moves* by process i, the total number of enabled guards in the processes $0..i$ is reduced.

Proof: By assumption, at least one of the processes $0..i$ has an enabled guard, and process $i + 1$ does not make an action. We will designate the three possible moves by process i as move 1, move 2, and move 3. The following cases exhaust all possibilities:

Case 1: If move 1 is of type (a), (b), (d) or (g), the result follows immediately.

Case 2: If move 1 is of type (e) or (f), after move 1, the condition $s[i] = s[i + 1]$ holds. In that case, move 2 must be of type (a), and the result follows immediately.

Case 3: If move 1 is of type (c), after the first move, the condition $s[i + 1]= s[i] + 1 \bmod 4$ holds. In this case, move 2 must be of types (b), (e), (f) or (g). If cases (b) and (g) apply, the result follows from case 1. If cases (e) and (f) apply, then we need a move 3 of type (a) as in Case 2, and the number of enabled guards in processes 0..i will be reduced. ■

Lemma 17.4 implies that if there is an enabled guard to the left of a process i, then after a finite number of moves by process i, a → appears at its right neighbor $i + 1$. Using similar arguments, one can demonstrate that if there is an enabled guard to the right of process $i(i > 0)$, then after a finite number of moves by process i, a ← appears at its left neighbor $i - 1$.

Lemma 17.5

In an infinite behavior, every process makes infinitely many moves.

Proof: Assume that this is not true, and there is a process j that does not make any move in an infinite behavior. By Lemma 17.4 and its follow-up arguments, in a finite number of moves, the number of enabled guards for every process $i \neq j$ will be reduced to 0. However, deadlock is impossible. So process j must make infinitely many moves.

Lemma 17.6

[Closure] If there is a single arrow, then all subsequent configurations contain a single arrow.

Proof: This follows from the cases (a), (e), (h), (k) in Table 17.1.
In an arbitrary initial state, there may be more than one → and/or ← in the system. To prove convergence, we need to demonstrate that in a bounded number of moves, the number of arrows is reduced to 1. ■

Lemma 17.7

[Convergence] Program *four-state* guarantees convergence to a legal configuration.

Proof: We argue that every arrow is eventually eliminated unless it is the only one in the system. We start with a →. A → is eliminated when it meets a ← (Table 17.1, case b) or another → (Table 17.1, case d). From Lemma 17.4, it follows that every → *eventually* moves to the right until it meets a ← or reaches process $n - 1$. In the first case, two arrows are eliminated. In the second case, the → is transformed into a ← after which the ← eventually moves to the left until it meets a → or a ←. In the first case (Table 17.1, case d), both arrows are eliminated, whereas in the second case (Table 17.1, case g), one arrow disappears and the ← is transformed into a →. Thus, the number of arrows progressively goes down. When the number of arrows is reduced to one, the system reaches a legal configuration. ■

Figure 17.4 illustrates a typical convergence scenario. Since closure follows from Lemma 17.6 and convergence follows from Lemma 17.7, the program four-state guarantees stabilization. This concludes the proof.

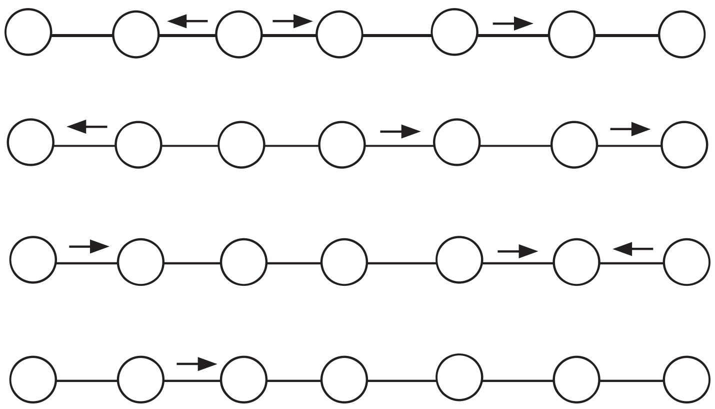

FIGURE 17.4 An illustration of convergence of the four-state algorithm.

17.4 STABILIZING GRAPH COLORING

Graph coloring is a classical problem in graph theory. Given an undirected graph $G = (V, E)$ and a set of colors C, node coloring defines a mapping from $V \rightarrow C$ such that no two adjacent nodes have the same color. In this section, we focus on designing a stabilizing algorithm for coloring the nodes of a *planar graph* using at the most *six colors*.

Section 10.4 illustrates a distributed algorithm for coloring the nodes of any *planar graph* with at the most *six* colors, but the algorithm is not stabilizing. In this section, we present the stabilizing version of it. Readers should review this algorithm before studying the stabilizing version.

The algorithm in Section 10.4.2 has two components. The first component transforms the given planar graph into a directed acyclic graph (dag) for which $\forall i \in V : outdegree(i) \leq 5$. The second component performs the actual coloring on this dag. Of the two components, the second one is stabilizing, since no initialization is necessary to produce a valid node coloring. However, the first one is not stabilizing, since it requires specific initialization (all edges were initialized to the state undirected). As a result, the composition of the two components is also not stabilizing. Our revised plan here has two goals:

1. Design a *stabilizing* algorithm A that transforms any planar graph into a dag for which the condition $P \equiv \forall i \in V : outdegree(i) \leq 5$ holds.
2. Use the dag-coloring algorithm B from Section 10.4 such that the desired postcondition Q reflecting a valid six-coloring holds.

If the actions of B do not negate any enabled guard of A, we can use the idea of *convergence stairs* [GM91] and run the two components concurrently to produce the desired coloring. We revisit algorithm B for coloring the dag. Recall that

- The color palette $C = \{0,1,2,3,4,5\}$
- $c(i)$ denotes the color of node i
- $succ(i)$ denotes the *successors* of a node i
- $sc(i) = \{c(j) : j \in succ(i)\}$