from [MRZ95]. The algorithms for sequential consistency are taken from Attiya and Welch [AW94]. For extension of these consistency conditions to multi-object operations see Garg and Raynal [GR99] and Mittal and Garg [MG98]. Causal consistency was introduced by Hutto and Ahamad [HA90].

Chapter 23

Self-Stabilization

Prudent, cautious self-control is wisdom's root — Robert Burns

23.1 Introduction

The algorithms for resource allocation problems that we have discussed so far do not work in the presence of faults. In this chapter we discuss a class of algorithms, called *self-stabilizing* algorithms, that can tolerate many kinds of faults.

We assume that the system states can be divided into legal and illegal states. The definition of the legal state is dependent on the application. Usually, system and algorithm designers are very careful about transitions from the legal states, but illegal states of the system are ignored. When a fault occurs, the system moves to an illegal state and if the system is not designed properly, it may continue to execute in illegal states. A system is called self-stabilizing if regardless of the initial state, the system is guaranteed to reach a legal state after a finite number of moves.

This chapter is organized as follows. Section 23.2 presents Dijkstra's self-stabilizing algorithm for mutual exclusion on a ring. This algorithm requires each machine on the ring to have at least as many states as the number of machines. Section 23.3 presents another mutual exclusion algorithm also due to Dijkstra, which requires only three states per machine.

The notation used in this chapter is summarized in Figure 23.1.

N	Number of processes
S	State of a process
K	Total number of states per machine
L	State of the left neighbor
R	State of the right neighbor
B	State of the bottom machine
T	State of the top machine

Figure 23.1: Notation

23.2 Mutual Exclusion with K-State Machines

We will model the mutual exclusion problem as follows. A machine can enter the critical section only if it has a *privilege*. Therefore, in the case of mutual exclusion, legal states are those global states in which exactly one machine has a privilege. The goal of the self-stabilizing algorithm is to determine who has the privilege and how the privileges move in the network.

Bottom:
if $(L = S)$ **then** $S := S + 1 \ mod \ K$;

For other machines:
if $(L \neq S)$ **then** $S := L$;

Figure 23.2: K-state self-stabilizing algorithm

We assume that there are N machines numbered $0 \ldots N-1$. The state of any machine is determined by its *label* from the set $\{0 \ldots K-1\}$. We use L, S, and R to denote the labels of the left neighbor, itself, and the right neighbor for any machine. Machine 0, also called the *bottom* machine, is treated differently from all other machines. The program is given in Figure 23.2, and a sample execution of the algorithm is shown in Figure 23.3. The bottom machine has a privilege if its label has the same value as its left neighbor, i.e., $(L = S)$. In Figure 23.3, the bottom machine and its left neighbor have labels 2 and therefore the bottom machine has a privilege. Once a machine possessing a privilege executes its critical section, it should execute the transition given by

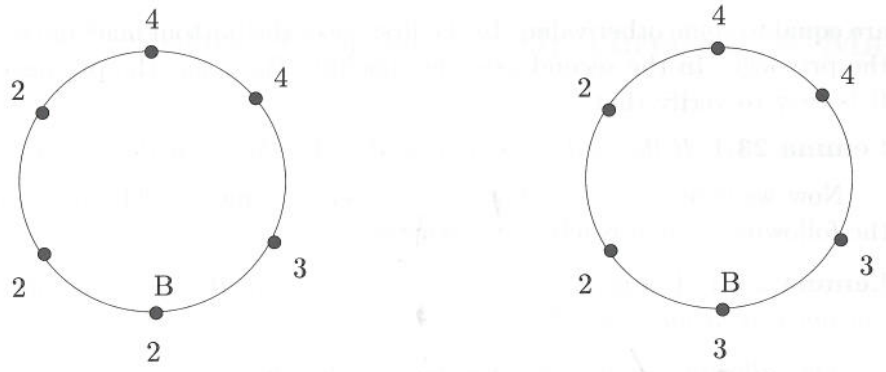

Figure 23.3: A move by the bottom machine in the K-state algorithm

the program. In Figure 23.3, on exiting from the critical section, the bottom machine executes the statement $S := S+1 \ mod \ K$ and acquires the label 3.

A normal machine has a privilege only when $L \neq S$. On exiting the critical section it executes $S := L$ and thus loses its privilege. In Figure 23.4, P_5 moves and makes it label as 4.

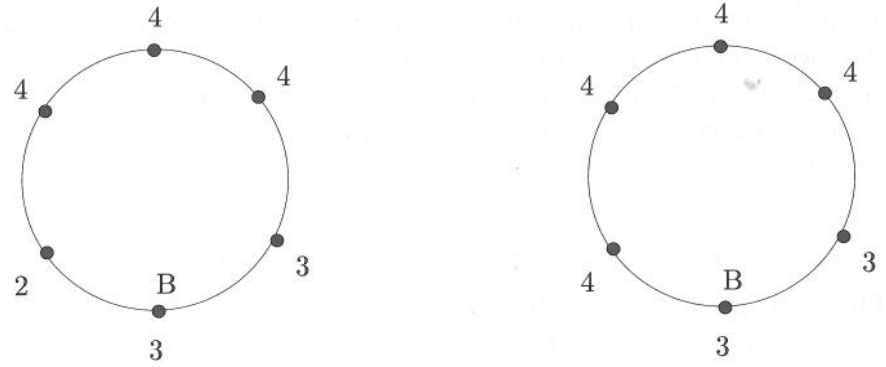

Figure 23.4: A move by a normal machine in the K-state algorithm

23.2.1 Proof of Correctness

In the above algorithm, the system is in a legal state if exactly one machine has the privilege. It is easy to verify that $(x_0, x_1, \ldots, x_{N-1})$ is legal if and only if either all $x_i's$ are equal or there exists $m < N-1$ such that all $x_i's$ with $i \leq m$ are equal to some value and all other $x_i's$

are equal to some other value. In the first case, the bottom machine has the privilege. In the second case the machine P_{m+1} has the privilege. It is easy to verify that

Lemma 23.1 *If the system is in a legal state, then it will stay legal.*

Now we consider any unbounded sequence of moves. The proof of the following lemma is left as an exercise.

Lemma 23.2 *A sequence of moves in which the bottom machine does not move is at most $O(N^2)$.*

The following lemma exploits the fact that $K \geq N$.

Lemma 23.3 *Given any configuration of the ring, either*
(1) no other machine has the same label as the bottom, or
(2) there exists a label that is different from all machines.
Furthermore, within a finite number of moves, (1) will be true.

Proof: We show that if (1) does not hold, then (2) is true. If there exists a machine that has the same label as that of bottom, then there are now $K-1$ labels left to be distributed among $N-2$ machines. Since $K \geq N$, we get that there is some label which is not used.

To show the second part, first note that if some label is missing from the network, then it can only be generated by the bottom machine. Moreover, the bottom machine simply cycles among all labels. Since, from Lemma 23.2, the bottom machine moves after some finite number of moves by normal machines, we get that the bottom machine will eventually get the missing label.

■

We now show that system reaches a legal state in $O(N^2)$ moves.

Theorem 23.4 *If the system is in illegal state, then within $O(N^2)$ moves, it reaches a legal state.*

Proof: It is easy to see that once the bottom machine gets the unique label, the system stabilizes in $O(N^2)$ moves.

The bottom machine can move at most N times before it acquires the missing label. Machine 1 therefore can move at most $N+1$ times before the bottom acquires the label. Similarly, machine i can move at most $N+i$ times before the bottom gets the label.

By adding up all the moves, we get
$N + (N+1) + \ldots + (N+N-1) = O(N^2)$ moves.

■

23.3 Mutual Exclusion with Three-State Machines

Bottom:
if $(B+1 = R)$ **then** $B := B+2$;

Normal:
if $(L = S+1)$ or $(R = S+1)$ **then** $S := S+1$;

Top:
if $(L = B)$ and $(T \neq B+1)$ **then** $T := B+1$;

Figure 23.5: Three-state self-stabilizing algorithm

The above algorithm requires that the number of states per machine K to be at least N. Therefore the algorithm is not independent of the number of machines. We now show another algorithm due to Dijkstra that requires only three states per machine independent of the total number of machines in the system. Assume that there is a ring of at least three machines. The program consists of three types of machines. In this configuration, we view machines as a sequence starting with a *bottom* machine, followed by one or more *normal* machines, and ending with a *top* machine. The state of any machine ranges over $\{0,1,2\}$. In the algorithm shown in Figure 23.5, all additions are performed modulo 3. A sample execution is shown in Figure 23.6.

23.3.1 Proof of Correctness

To prove correctness of the algorithm, it is useful to view the state of the system as a string starting with B, then states of the normal machines S's, and ending with T. Between any two consecutive states, there are only two possibilities—either they are the same or they differ by 1. In case they differ by 1, we put an arrow between the state such that arrow points to the direction in which the number decreases by 1 modulo 3. Thus 2 points to 1, 1 points to 0, and 0 points to 2.

Our proof uses the following three different metrics based on the

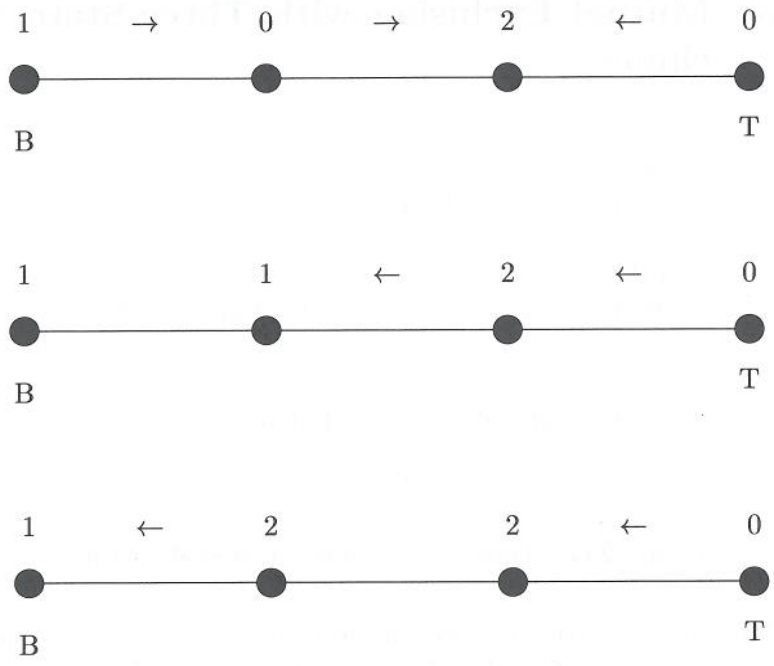

Figure 23.6: A sample execution of the Three-state self-stabilizing algorithm

arrows present in the string.

x = the number of arrows

y = the number of left-pointing arrows + twice the number of right-pointing arrows

$$z = \sum_{\substack{\text{left-pointing}\\ \text{arrows}}} \text{distance from bottom} + \sum_{\substack{\text{right-pointing}\\ \text{arrows}}} \text{distance from top}$$

The three-state machine algorithm can be rewritten as:

For bottom:

(0) $B \leftarrow R$ to $B \rightarrow R$ $\quad \Delta x = 0 \quad \Delta y = +1 \quad \Delta z = N - 1$

For a normal machine:

(1) $L \rightarrow S \quad R$ to $L \quad S \rightarrow R$ $\quad \Delta x = 0 \quad \Delta y = 0 \quad \Delta z = -1$

(2) $L \quad S \leftarrow R$ to $L \leftarrow S \quad R$ $\quad \Delta x = 0 \quad \Delta y = 0 \quad \Delta z = -1$

(3) $L \rightarrow S \leftarrow R$ to $L \quad S \quad R$ $\quad \Delta x = -2 \quad \Delta y = -3 \quad \Delta z = -N + 1$

(4) $L \rightarrow S \rightarrow R$ to $L \quad S \leftarrow R$ $\quad \Delta x = -1 \quad \Delta y = -3 \quad \Delta z \leq N - 2$

(5) $L \leftarrow S \leftarrow R$ to $L \rightarrow S \quad R$ $\quad \Delta x = -1 \quad \Delta y = 0 \quad \Delta z \leq N - 2$

For top:

(6) $L \rightarrow T$ to $L \leftarrow T$ $\quad \Delta x = 0 \quad \Delta y = -1 \quad \Delta z = N - 1$

(7) $L \quad T$ to $L \leftarrow T$ $\quad \Delta x = +1 \quad \Delta y = +1 \quad \Delta z = N - 1$

It is easy to see that if a string has a single arrow then the system is in a legal state. Furthermore, the arrow remains the only one. It moves left to right using moves (1) and (2). It gets reflected by the bottom using the move (0) and by the top using the move (6). The move (6) is possible because if there is a single arrow, the precondition $(L = B)$ and $(T \neq B + 1)$ holds.

Moreover, if the system does not have any arrows, then move (7) is the only eligible move and it will create a single arrow. It also follows that at least one move is always possible in the system. Our goal is to bound the number of moves before the string gets into a legal state.

Let m_i denote the number of moves corresponding to the transformation (i), where $i \in [0 \ldots 7]$, made by the system before it stabilizes.

Lemma 23.5 *Between two successive moves of top at least one move of bottom takes place.*

Proof: The top machine moves only when $T \neq B + 1$. When it moves, it makes T equal to $B + 1$. Now the top machine can move only when B changes.

■

Lemma 23.6 *A sequence of moves in which bottom does not move is finite.*

Proof: From Lemma 23.5, it is sufficient to consider the moves of the normal machines. Moves (3),(4), and (5) decrease the number of arrows, and no moves of the normal machines increase the number of arrows; therefore, it is sufficient to show that a sequence of moves (1) and (2) is finite. However, this follows from the structure and the finiteness of the string. The move (1) takes the arrow from left to right, and therefore the arrow must eventually hit the top machine. A similar argument applies to move (2).

■

Lemma 23.7 *If the string has not reached a legal state, then y decreases by at least 1 per move of bottom.*

Proof: Between successive moves of bottom, falsification of "leftmost arrow exists and points to the right" happens in (3), (4), or (6). If this happens in move (6), then the string is legal and we are done. If the falsification happens in (3) or (4), then y decreases by 3. But y can increase by at most 2 on account of moves (0) and (7) between two successive moves of bottom, Therefore, we get that y decreases by at least 1 per move of bottom.

■

Theorem 23.8 *Within $O(N^2)$ moves, there is one arrow in the string.*

Proof: From Lemma 23.7, if the algorithm has not yet reached a legal state, then y decreases by at least 1 per move of bottom. As the initial value of y is at most $2N$,

$$m_0 \leq 2N.$$

Also, from Lemma 23.5, between two successive moves of top at least one move of bottom takes place. Therefore,

$$m_6 + m_7 \leq 2N.$$

We now bound the number of moves made by normal machines. Note that the transformations (3), (4), and (5) decrease x whereas the transformations (0), (1), and (2) do not change x at all. Furthermore, a move by top increases x by at most 1 and the total number of such moves is upper-bounded by $2N$. Since the maximum initial value of x is N,

$$m_3 + m_4 + m_5 \leq 3N.$$

Finally, since the transformations (1) and (2) decrease z, the other transformations—which are at most $7N$ in total—increase it by at most N and the maximum initial value of z is N^2,

$$m_1 + m_2 \leq 8N^2$$

giving us an upper bound of $O(N^2)$ on the total number of moves required for the system to stabilize.

■

The following example establishes that it is, in fact, a tight upper bound. Consider the string consisting of $(N-1)/2$ right-pointing arrows followed by $(N-1)/2$ spaces, for some odd $N > 3$. By repeatedly using transformation (1) $(N-1)^2/4$ times, we obtain the string consisting of $(N-1)/2$ spaces followed by $(N-1)/2$ right-pointing arrows; each arrow moves to the right by $(N-1)/2$ places, with the system staying in an illegal state throughout.

23.4 Problems

23.1. Show that a system with four machines may not stabilize if it uses the K-state machine algorithm with $K = 2$.

23.2. Prove Lemma 23.2.

23.3. Show that the K-state machine algorithm converges to a legal state in at most $O(N^2)$ moves by providing a norm function on the configuration of the ring that is at most $O(N^2)$, decreases by at least 1 for each move, and is always nonnegative.

23.4. In our K-state machine algorithm we have assumed that a machine can *read* the value of the state of its left machine and *write* its own state in one atomic action. Give a self-stabilizing algorithm in which a processor can only *read* a remote value or *write* a local value in one step, but not both.

*23.5. **[due to [Dij74]]** Show that the four-state machine algorithm in Figure 23.7 is self-stabilizing. The state of each machine is represented by two booleans xS and upS. For the bottom machine $upS = true$ and for the top machine $upS = false$ always hold.

Bottom:
if $(xS = xR)$ and $\neg upR$ then $xS := \neg xS$;

Normal:
if $xS \neq xL$ then $xS := \neg xS; upS := true$;
if $xS = xR$ and upS and $\neg upR$ then $upS := false$;

Top:
if $(xS \neq xL)$ then $xS := \neg xS$;

Figure 23.7: Four-state self-stabilizing algorithm

23.6. A self-stabilizing algorithm is *uniform* if all processes are equal and do not have any process identifiers. Show that there is no self-stabilizing algorithm for mutual exclusion in a ring with a nonprime number of processes.

*23.7. Assume that each process P_i has a pointer that is either null or points to one of its neighbors. Give a self-stabilizing, distributed algorithm on a network of processes that guarantees that the system reaches a configuration where (1) if P_i points to P_j then P_j points to P_i, and (2) there are no two neighboring processes such that both have null pointers.

23.5 Bibliographic Remarks

The idea of self-stabilizing algorithms first appeared in Dijkstra [Dij74], where three self-stabilizing algorithms were presented for mutual exclusion in a ring. The proof for the Three-state algorithm is taken from Mittal and Garg [MG01b].

Chapter 24

Knowledge and Common Knowledge

Imagination is more important than knowledge. — Albert Einstein

24.1 Introduction

Many problems in a distributed system arise from the lack of global knowledge. By sending and receiving messages, processes increase the knowledge they have about the system. However, there is a limit to the level of knowledge that can be attained. In this chapter, we use the notion of knowledge to prove some fundamental results about distributed systems. In particular, we show that agreement is impossible to achieve in an asynchronous system in the absence of reliable communication.

The notion of knowledge is also useful in proving lower bounds on the message complexity of distributed algorithms. In particular, knowledge about remote processes can be gained in an asynchronous distributed system only by message transfers. For example, consider the mutual exclusion problem. It is clear that if process P_i enters the critical section and later process P_j enters the critical section, then there must be some knowledge gained by process P_j before it can begin eating. This gain of knowledge can happen only through a message transfer. Observe that our assumption of asynchrony is crucial in requiring the message transfer. In a synchronous system with a global clock, the knowledge can indeed be gained simply by passage of time. Thus for a mutual exclusion algorithm, one may have time-division multiplexing in which processes enter the critical section on their pre-