21

MASSACHUSETTS
COMPUTER ASSOCIATES, INC.

~~LAKESIDE OFFICE PARK, WAKEFIELD, MASS. 01880 • 617/245-9540~~

26 PRINCESS STREET, WAKEFIELD, MASS. 01880 • 617 / 245-9540

On Self-stabilizing Systems

by

Leslie Lamport

December 5, 1974
CA 7412-0511

A SUBSIDIARY OF APPLIED DATA RESEARCH, INC.

Introduction

Dijkstra has recently described a type of formal system consisting of a network of interconnected machines [1]. The next state of each machine is a function of its state and the states of its neighbors. At any instant, the state of the system is described by the states of all the machines. The system is assumed to have a normal mode of operation in which its state is always a "legitimate" one. (This will all be defined more precisely below.) The system is called self-stabilizing if it will eventually enter its normal mode of operation when started in any initial state.

These formal systems are of interest for modeling networks of independent processors. (To model the delay in transmitting information between processors, the transmission line can be represented by a machine.) Self-stabilizing systems represent ones which are self-correcting: even if they reach an incorrect state through some transient malfunction, they will eventually resume correct operation.

Dijkstra described some self-stabilizing systems in which the machines are connected in a ring and in a line. These systems had the property that in normal operation, exactly one machine could change its state at any time. This is a useful property because it means that the network of autonomous machines has been synchronized so that it cycles through a fixed sequence of states in which only one machine at a time has a "privileged status". Several such systems could also be combined to form a self-stabilizing system in which several machines could have different privileges at the same time.

In this paper, we construct such a self-stabilizing system for a network described by any arbitrary connected graph. We also consider the solution to the mutual exclusion problem presented in [2], and show that it can be implemented as a self-stabilizing system. Our definitions will also include some useful generalizations of the concepts introduced by Dijkstra.

Definitions

A system consists of a set of machines which form the nodes of an undirected graph. If there is an arc between two machines, then they are said to be neighbors. The state of the system at any time consists of the states of each machine. Every machine has a set of privileges, each of which is a boolean function of the state of that machine and the states of its neighbors. The privilege is said to be present in a system state if it has the value true. Associated with each privilege is a move, which defines a next state of the machine as a function of the current states of it and its neighbors.

The system advances to its next state by the following sequence of operations, which form a system step.

(1) Choose any non-empty subset of all the privileges which are now present, containing at most one privilege from each machine. (If no privileges are present, the system has halted.)

(2) For each privilege in this subset, use its associated move to determine the next state of that machine.

(3) All of these machines are then simultaneously changed to their new states.

We assume that the system has a set of <u>legitimate</u> system states. The system is said to be <u>live</u> if it has the following three properties:

(1) From each legitimate state, any system step leaves the system in a legitimate state.

(2) For any pair of legitimate states, there exists a possible sequence of system steps leading from the first state to the second.

(3) Each privilege is present in some system state.

The system is called <u>self-stabilizing</u> if it can never halt, and there exists a number N such that if the system is started in any initial state then after N system steps it will be in a legitimate state.

Our definitions are essentially the same as Dijkstra's except for some small generalizations: (i) we allow infinite-state machines, (ii) we use a "distributed daemon" instead of his "central daemon", and (iii) Dijkstra only considered live systems. We now make two new definitions.

In multiprocessor systems, one usually assumes that no processor can be infinitely faster than another. This assumption has its analogue in the following additional assumption about the system's operation: there exists a number N such that a privilege cannot be present in N successive system states without being chosen during substep (1) of one of the intervening system steps. A system which is self-stabilizing under this extra assumption is said to be <u>weakly self-stabilizing</u>.

It is sometimes desirable to weaken property (2) of live systems in order to ignore inessential differences between system states. Let us call two system states <u>equivalent</u> if any privilege is present in one if and only if it is present in the other. A <u>semi-live</u> system is then defined to be one which satisfies properties (1) and (3) of live systems and the following property: (2') For any pair of legitimate states, there exists a possible sequence of system steps leading from the first to a state equivalent to the second.

<u>Self-Stabilizing Live Systems with Arbitrary Graphs</u>

Given an arbitrary connected, undirected graph, we now construct a self-stabilizing live system for this graph -- i.e., a system with a machine for each node such that two machines are neighbors only if there is an arc between the corresponding nodes. In each legitimate system state, exactly one privilege is present. The number of states of each machine is less than or equal to twice the number of neighbors it has. The system is a generalization of a slightly altered version of Dijkstra's "four-state machine" systems.

By deleting arcs, we can make the graph a connected acyclic graph. (If the arc between two machines is deleted, then those machines are trivial neighbors which do not actually affect each other. A procedure which makes all directly connected machines into non-trivial neighbors is described later.) Hence, we can assume that the given graph is acyclic. We can then make the graph into a tree by choosing a root node, and defining a father/son relation among nodes in the obvious way so that the root node is an ancestor of all other nodes.

The state of each machine has two components: a color and a pointing state. The color can assume either of two values. The pointing state defines which of the machine's neighbors it is <u>pointing</u> at. An arbitrary cyclic ordering of a machine's neighbors is assumed, so the <u>next</u> neighbor (next after the one it is pointing at) is defined. The root node is assumed to have some arbitrarily chosen son defined as its <u>number one son</u>.

Every machine M has one privilege for each neighbor N . The privilege is present if and only if M and N both point at each other and either (i) N is M's son and they have different colors, or (ii) N is M's father and they have the same color. For each privilege, the move is the following:

(1) Make M point at its next neighbor.

(2) <u>If</u> (after step (1)) M now points at its father, or M is the root node and it now points at its number one son, <u>then</u> change M's color.

Note that if M is a leaf node, or if it is the root node and has only one son, then a move simply changes its color.

A non-root node is defined to be at rest if it and all its descendants have the same color, and they all point to their fathers. The definition is the same for a root node, except that the root node itself must point to its number one son. A rest state of the system is one in which the root is at rest. (There are two such states.) The legitimate states are defined to be those which can be reached when the system is started in a rest state.

The fact that the system is live, and that each legitimate state has only one privilege present, is easily proved by induction on the height of the tree. We now sketch a proof that the system is self-stabilizing. First, observe that a privilege must be present whenever two nodes point at one another. Since the root node points to a son and each leaf node points to its father, it is clear that the system can never halt. Now consider any infinite sequence of successive system states. Some machine must move infinitely many times during that sequence of steps. But that is possible only if each of its neighbors changes color an infinite number of times. Hence, all machines move infinitely often in the sequence. This proves that starting in any initial state, each machine must eventually reach each of its states. Using this result, a straightforward induction argument proves that for each node and any initial state, the system must eventually reach a state in which that node is at rest -- thus proving self-stabilization.

In constructing this system, we deleted some arcs if the original graph was cyclic. To avoid separating neighbors, we can instead obtain an acyclic graph by splitting nodes. After constructing the above system for this graph, we recombine the split nodes by merging their machines in the obvious way.

The Bakery Algorithm

In [2] we presented a "bakery algorithm" for solving the mutual exclusion problem. We assume that the reader is familiar with this algorithm. It has the following self-stabilizing property. Suppose that a process cannot remain forever in its non-critical section, and that each processor is started at any point in its program with any non-negative value of <u>number</u>[i]. Then, the system will eventually assume its normal mode of operation. To see this, observe that if process i has entered and left the doorway at least once, then Assertions 1 and 2 of [2] will hold for all k. If process i has the smallest value of (<u>number</u>[i], i) then it will eventually enter the doorway, choose a new value of <u>number</u>[i] greater than any initial value of <u>number</u>[j] which has not been changed, and leave the doorway. Hence, all processors will eventually pass through the doorway. After this has happened, Assertions 1 and 2 will hold, so the system will be operating correctly.

The above discussion was in terms of the notation and assumptions of [2]. We now consider implementation of the bakery algorithm by our systems of interconnected machines. We will see that a large class of "natural" implementations are weakly self-stabilizing, semi-live systems. A more restricted class of implementations are self-stabilizing live systems. For the sake of brevity, the discussion will be informal, and no proofs will be given.

Since interprocessor communication occurs only by one processor reading another processor's memory, the algorithm is easy to implement with our systems of machines. The Algol program for each processor can be directly translated into a machine whose state is defined by the value of a "program counter" and the contents of certain "memory registers". We allow the execution of a single statement to be represented by several program steps, and allow memory registers to hold the results of intermediate calculations as well as the values of program variables. All privileges will be boolean expressions of the form "program counter = x", except for pairs of privileges of the form "program counter = x **and** f" and "program counter = x **and not** f" which come from an **if** statement whose conditional expression f is a function of other processors' variables.

Define a proper state of the system to be one which can be reached from the normal initial state (the one with each processor in its non-critical section, etc.). The behavior of the algorithm is essentially unchanged if at some instant one changes some of the non-zero elements of the array number in a way which does not change the numerical ordering relations among the elements. Roughly speaking, we define a semi-proper system state to be one obtained from a proper state by such a change. The exact definition is complicated by considering the intermediate result registers, and is left to the reader.

We first define the legitimate states of the system to consist of all proper and semi-proper states. This defines a semi-live system. It is not live because the system cannot go from a proper state to a semi-proper one. The system will be weakly self-stabilizing if the implementation obeys the following two rules:

(1) The value of j must always lie between 1 and N (or be interpreted as a number in that range). Alternatively, we can eliminate the variable j by expanding the for loop.

(2) Machine i does not have states in which choosing[i] has the incorrect value. Equivalently, choosing[i] can be eliminated and its value inferred by reading the value of i's program counter.

Such an implementation is weakly self-stabilizing, but it is not self-stabilizing because the following types of system behavior must be disallowed.

(i) One machine loops forever at statement L2 or L3 while no other machine moves.

(ii) Some machine i remains in its critical section forever with number[i] equal to zero.

We obtain a self-stabilizing system by placing the following conditions on the implementation.

(i) Each waiting loop at L2 or L3 is implemented by a single privilege which is present if and only if the loop exiting condition is true.

(ii) We only include machine states in which number[i] is zero if and only if it should be zero.

Suppose it were true that from a semi-proper initial state the system must eventually reach a proper state. Then these implementations would be live systems if the legitimate states were defined to be the proper states. We do not know if this is true for all of these implementations, but suspect that it is not. However, it is true if for each machine, the statement in the doorway which assigns a value of number[i] is executed by a single program step. (The proper states then have the property that the non-zero elements of the array number form a set of consecutive integers.) Hence, such implementations produce live systems.

These systems contain infinite-state machines, since the values of number[i] can become arbitrarily large. There is a complicated modified version of the bakery algorithm in which the values of all variables are bounded. This algorithm has the same self-stabilization properties as the ordinary bakery algorithm.

References

[1] Dijkstra, E.W. Self-stabilizing Systems in Spite of Distributed Control. Comm. ACM 17, 11 (November, 1974), 643-644.

[2] Lamport, L. A New Solution of Dijkstra's Concurrent Programming Problem. Comm. ACM 17, 8 (August, 1974), 453-455.