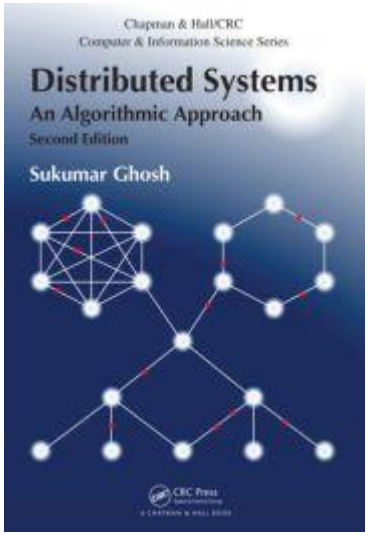
Self-Stabilizing Systems



Nonmasking fault tolerance

Self-stabilization

In presence of failures, a robust algorithm guarantees:

- Liveness properties are eventually achieved
- Safety properties are never violated

Self-stabilization is a different approach to fault tolerance

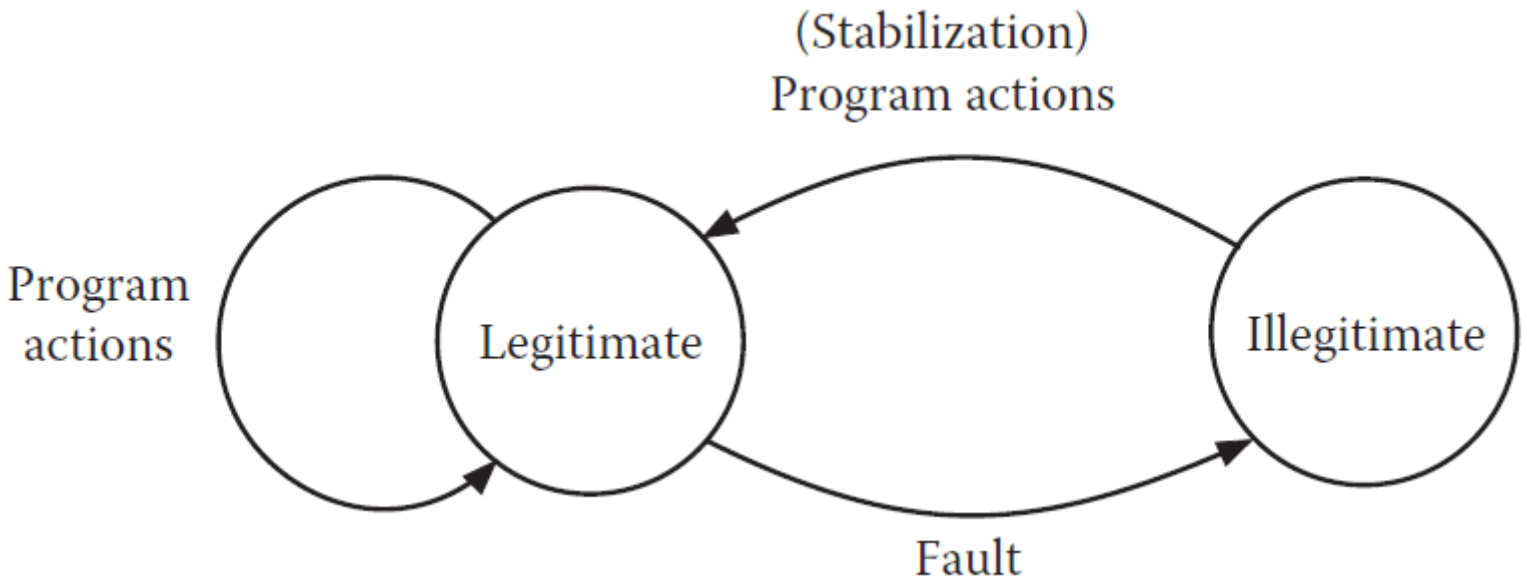
- It considers transient (temporary) failures
- It is optimistic, if safety is violated, the system will recover within a finite time, and will behave nicely afterwards

States in a self-stabilizing system

A system is called self-stabilizing when the following two conditions hold:

- Convergence
  - ✓ Regardless of the initial state, and regardless of elegible actions chosen for execution at each step, the system eventually returns to a legitimate state
- Closure
  - ✓ Once in a legitimate state, the system continues to be in a legitimate state unless a fault corrupts the data memory

States in a self-stabilizing system



A legitimate state is usually represented by an **invariant** over the global state of the system

Self-stabilizing Systems in Spite of Distributed Control

Dijkstra, E.W. (1974).
Communications of the ACM, Vol. 17, No. 11, pp. 643-644.

EWD 391

Self-stabilization

Self-stabilizing systems with distributed control do exist in the sense that **local decisions** force the system towards satisfying and then maintaining a **global predicate**

Dijkstra demonstrated its feasibility by proposing three self-stabilizing solutions to the problem of distributed **mutual exclusion** for a ring with N+1 machines (0 .. N) each having K states

K-state self-stabilizing algorithm

Unidirectional ring of K-state machines (K>N)

Bottom:
  if (S=L) then S:=S+1 mod K
For other machines:
  if (S≠L) then S:=L

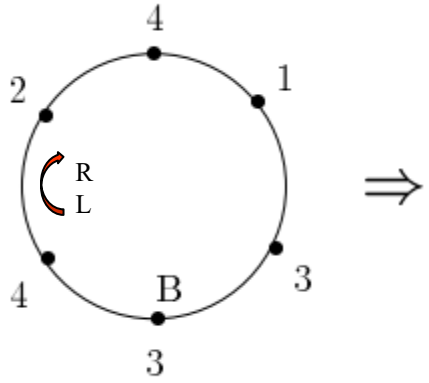
Each machine state is characterized by a variable ranging over {0,1, … K-1}

Dijkstra

- defined a privilege of a machine to be the ability to change its current state (“move”)
- considered a legitimate state as one in which exactly one machine enjoys the privilege

K-state self-stabilizing algorithm

Example



Bottom:
  if (S=L) then S:=S+1 mod K
For other machines:
  if (S≠L) then S:=L

B
3 4 2 4 1 3

3 4 2 4 4 **3**
4 4 **2 4** 4 **3**
4 4 4 4 4 **3**
4 4 4 4 4 4
5 **4** 4 4 4 4
5 5 **4** 4 4 4
5 5 5 **4** 4 4
5 5 5 5 **4** 4

convergence

closure

K-state self-stabilizing algorithm

State variable

- Process i has a counter S(i) that takes values in the range [0, K-1].
- Its initial value is arbitrarily chosen from [0, K-1].

K-state self-stabilizing algorithm

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

- If every process 1 through N has a disabled guard (S=L) , then for all i>0 S[i]=s[i-1].
- But this implies S[0]=S[N], so process 0 has an enabled guard.

K-state self-stabilizing algorithm

Convergence. Starting from any illegal configuration (m>1 enabled guards), the ring eventually converges to a legal configuration (1 enable guard).

One important fact used in the proof is that the nonzero processes do not contain all the possible counter values. By this observation, together with the fact that each process is infinitely often enabled, we can prove that each infinite run of the algorithm will reach a legitimate state.

K-state self-stabilizing algorithm

$B=S_0$ executes its action infinitely often.

Suppose that B is not privileged, i.e., $S[0] \neq S[N]$, then in a finite number of commands it will become privileged.

Let $1 \leq j \leq N$ be the minimum value such that $S[j] \neq S[0]$; because j is the minimum value, $S[j-1]=S[0]$; and therefore $S[j] \neq S[j-1]$, i.e., machine S_j is privileged.

K-state self-stabilizing algorithm

In a finite number of commands, the demon will point to j, thus increasing its counter if j<N or making S[N]=S[0] if j=N, i.e., making S_0 privileged.

So, B will continue forever to get the opportunity to fire.

K-state self-stabilizing algorithm

There are N+1 process with K (K>N) states per process.

By the pigeonhole principle, at least one integer j must not be the initial state of any process i>0.

Each action by process i>0 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.

K-state self-stabilizing algorithm

However, it is guaranteed that at some point, S[0]=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.

K-state self-stabilizing algorithm

Closure. The legal configurations satisfy the closure property. A single privilege will rotate past all machines.

K-state self-stabilizing algorithm

If only process 0 has an enabled guard, then

- S[0] = S[N]
- For all i>0 S[i]=S[i-1].

A move by process 0 will disable its own guard and enable the guard of process 1.

K-state self-stabilizing algorithm

If only process i (0<i<N) has an enabled guard, then

- S[i]≠S[i-1]
- For all j<i: S[j]=S[i-1]
- For all j>i: S[j]=S[i]
- S[0] ≠ S[N]

A move by process i will disable its own guard and enable the guard for process i+1.

K-state self-stabilizing algorithm

If only process N has an enabled guard, then

- $S[N] \neq S[N-1]$
- For all $0<i<N$: $S[i]=S[i-1]$
- $S[0] \neq S[N]$

A move by process N will disable its own guard and enable the guard for process 0.

Once in a legitimate state, the system will remain in legitimate states.

A Belated Proof of Self-stabilization

Dijkstra, E.W. (1986).
Distributed Computing, Vol. 1, No. 1, pp. 5-6.

EWD 922

Three-state self-stabilizing algorithm

Bidirectional ring of at least 3 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** $T := B+1;$

Each machine state is characterized by a variable ranging over {0,1,2}

Additive operations are understood to be reduced mod 3

The privilege is not required to rotate

Three-state self-stabilizing algorithm

The ring consists “in order from left to right” of

B S_1 ... S_{N-1} T

The configuration is viewed as a string of 0,1,2

Between neighbours whose states differ, we place an arrow such that in the direction of the arrow the state decreases (mod 3) by 1

Dijkstra defined $y = \#\leftarrow + 2\#\rightarrow$

Three-state self-stabilizing algorithm

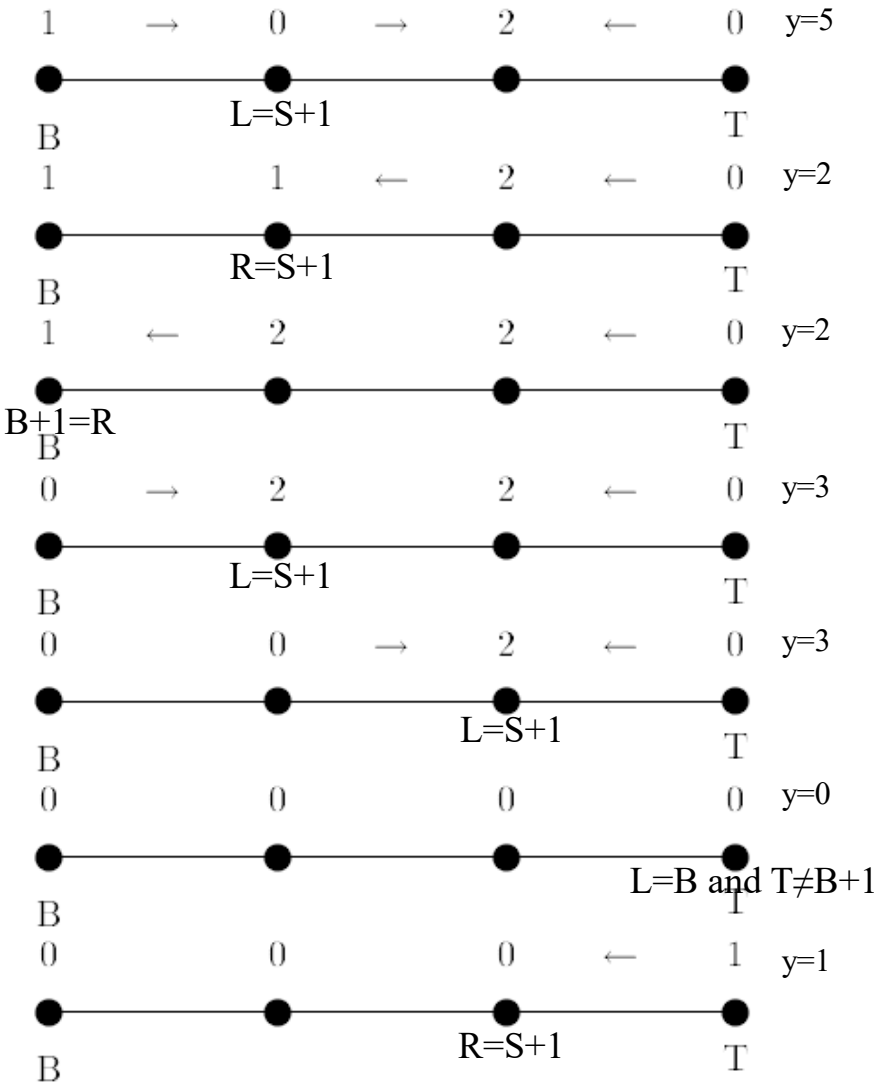
Example of convergence

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** $T := B+1$;

$y = \#\leftarrow + \ 2\#\rightarrow$



Three-state self-stabilizing algorithm

Bottom :

* (0) B $\leftarrow$ R to B $\rightarrow$ R $\Delta y = +1$

Normal Machine:

* (1) L $\rightarrow$ S R to L S $\rightarrow$ R $\Delta y = 0$

* (2) L S $\leftarrow$ R to L $\leftarrow$ S R $\Delta y = 0$

(3) L $\rightarrow$ S $\leftarrow$ R to L S R $\Delta y = -3$

(4) L $\rightarrow$ S $\rightarrow$ R to L S $\leftarrow$ R $\Delta y = -3$

(5) L $\leftarrow$ S $\leftarrow$ R to L $\rightarrow$ S R $\Delta y = 0$

Top Machine (privilege also depends on B):

* (6) L $\rightarrow$ T to L $\leftarrow$ T $\Delta y = +1$

* (7) L T to L $\leftarrow$ T $\Delta y = +1$

Three-state self-stabilizing algorithm

No deadlock. Under all circumstances at least one move is posible.

Proof:

There are 9 possible strings L{“ ”,→,←}S{“ ”,→,←}R

All of them are covered by transitions (0) to (7)

Three-state self-stabilizing algorithm

If the string is free of arrows, L=B and T=B and transition (7) creates the situation of precisely 1 arrow in the string

If the string contains 1 arrow, that one arrow remains the only one, travelling up and down the string via transitions (1) and (2), and being reflected at the ends via transitions (0) and (6)

If the string contains multiple arrows, y is decreased within a finite number of moves (yielding a string with 0 or 1 arrow)

?!

Three-state self-stabilizing algorithm

From the design of the move of Top

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

we conclude

Lemma 0: Between two successive moves of Top at least one move of Bottom takes place

Three-state self-stabilizing algorithm

Lemma 1: A sequence of moves in which Bottom does not move is finite

Proof:

- Any sequence of (only) normal (machine) moves is finite
  - From the topology and the finite length of the string, the sequence of transitions (1) and (2) is finite
  - Transitions (3), (4) and (5) decrease the number of arrows, and none of the normal moves increases that number
- From Lemma 0, in a sequence of moves in which Bottom does not move, at most one move of Top occurs

Three-state self-stabilizing algorithm

Theorem (convergence): Within a finite number of moves there is one arrow in the string

Proof:

- From Lemma 1, in an unbounded sequence of moves the number of moves of Bottom is unbounded
- Between two successive moves of Bottom, the situation “the left-most arrow (exists and) points to the right” is falsified at least once by transitions (3), (4), or (6)

Three-state self-stabilizing algorithm

Proof (continuation):

- if transition (6)
  - we have one arrow in the string and we are done
- else // transitions (3) or (4)
  - y is decreased by 3; and since only transitions (0) and (7) increase y, but (by lemma 0) by at most 2 per move of Bottom (+1 of Bottom and +1 of Top), the net result is that, per move of Bottom, y is decreased by at least 1

An alternative solution to a problem on self-stabilization

Ghosh, S. (1993).
ACM Transactions on Programming Languages and Systems, Vol. 15, No. 7, pp. 327-336.

Four-state self-stabilizing algorithm