1 Introduction

1.1 The Problem Domain

Today, we are going to ask whether a system can recover from perturbation.

Consider a children's top: If it is perfectly vertically, you can nudge it into precession, and it will sometimes recover[1].

The formal consideration of this question begin in operations research, where the systems were industrial/-commercial — machines and manufacturing proceesses. **TODO 1:** *Back this up with a citation from the self-stabiliation survey that asserts this.*

In computer science, Dijkstra is the first to consider this problem in 1974. Let's start with an informal definition.

Definition 1.1. A distributed system is *self-stabilizing* if, from an "illegitimate" state, it returns to a "legitimate" state in finite time, without external (e.g. human) intervention.

To unravel this definition, we first need to answer three questions:

1. What do we mean by distributed system?
2. How do we model the distributed system?
3. What are legitimate and illegitimate states?

1.1.1 Distributed System

A distributed system runs an algorithm that communicates and, thereby, implements a protocol.

Below, I don't distinguish between the system and the algorithm that it runs, and thus use system and algorithm interchangeably.

1.1.2 Model

As usual, we model the system as a graph $G = (V, E)$, as depicted in Figure 1.

Each node in V executes. You can think of a node as a process or as a system, as you like. Each node has local state.

In his seminal paper, Dijkstra restricted each node to run an FSM. This is overly restrictive, as there is no reason other than the space required to encode its tape, not to imagine that each node is a TM.

In any event, we restrict our attention to the local states defined by the distributed algorithm under consideration.

The edges in E represent communication either via shared memory or message channels.

For the distributed system $G = (V, E)$ and $i, j \in V$, let

[1] See `http://www.4physics.com/phy_demo/top/top.html`, which is backed up in Bib.

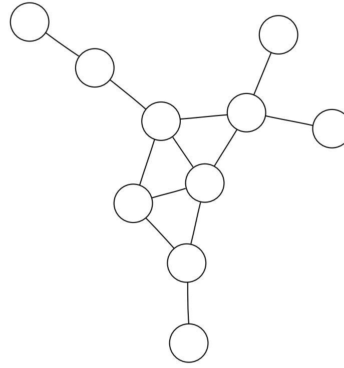

Figure 1: A Distributed System

1. s_i be the local state of i; and

2. q_{ij} be the contents of either a message queue from i to j or register shared by i and j which i can both read and write, but j can only read.

Note: Dolev's shared register definition is vague: "Processors may write in a set of registers and may read from a possibly different set of registers."

We aggregate the local states and messages into a vector, or *configuration*, to form a global, system state:

$$< s_0, s_1, \cdots, s_{n-1}, q_{ij} >$$

Self-stabilization is defined in terms of global, system state. From this point forward, when we use state, we mean global, not local, state.

These global states themselves form a graph, *distinct from the system graph*, where the edges are possible moves in the distributed algorithm.

Draw an example state graph.

1.1.3 Legitimate vs. Illegitimate States

The distributed application partitions the global state space into legitimate and illegitimate states.

Informally, legitimate states are those that we want the system/program to stay in and illegitimate states are all other possible states.

Partition state graph.

In most programs, the legitimates states are a small subset of the state space, and our goal is to contain the program within the legitimate subset, as Listing 1 illustrates.

Listing 1: "|Legitimate States| << |Illegitimate States|"

```
int x, y, z; // Assume unbounded registers, i.e. ignore overflow
z = x + y;   // There is only one legitimate state.
```

1.2 Termination

A self-stabilizing system $S = (V, E)$ never terminates.

We are only interested in whether or not the system global terminates, not whether or not some local machines

Definitionally, we are only interested in distributed, by which we mean communicating, systems. If a distributed system globally terminates, as distinct from the local termination of a single node within the system, it no longer communicates.

A self-stabilizing system $S = (V, E)$ never terminates. Assume, to the contrary, that every node in V has a terminal state s_t. Let c be the configuration, or global state, in which every node is in s_t. By definition of s_t, c is terminal in the global system graph. Since S is self-stabilizing, c must be legitimate, but, in c, no node communicates with any other. S is therefore not a distributed system, thus not self-stabilizing.

1.3 The Adversary

The self-stabilization adversary controls scheduling and can

1. Inject faults at an interval long enough for recovery; or, in other words,
2. Drop the system into any arbitrary global state.

In terms of the second formulation, self-stabilizing systems are those that do not require initialization.

In constrast with a Byzantine adversary, a self-stabilizing adversary is

- Not able to corrupt and change the state transitions, or program, of the machine, running at some node;
- Not persistent in time intervals less than recovery time; and
- Not adaptive, to the extent to which adaption requires unconstrained persistence.

Self-stabilization models transient faults, those that change state (local and therefore global), but *not* behavior which we assume is inviolate.

Ask the class "What criticisms of this model come to mind?"

Others have questioned the following

1. The assumption that program code, or the transition of the machine at each node, is inviolate.

2. Transient faults as opposed to persistent, continuous faults.
3. Since a self-stabilizing program may find itself in an illegitimate state, it may produce incorrect output. This must be tolerated. Would self-stabilization have prevented the Therac disaster?

Anecdotal aside: In my ten years in industry, as a system administrator and programmer, I never faced a Byzantine adversary (at least not one that I was aware of ;-)), but spent a lot of time wrestling self-stabilizing ones.

1.4 Cleaned up Dijkstra definition

Every node in the system graph has an associated privilege. The scheduler can select or "move" only those nodes whose privilege is true, or *present*.

Definition 1.2. A *privilege* is a boolean function over the states of a node and its neighbors in the system graph.

A “Legitimate” legitimate set of global states, or configurations, must obey the following constraints:

1. In each legitimate state, one or more privileges are present;
2. In the absence of faults, it is impossible to move from a legitimate state to an illegitimate state;
3. Each privilege is present in at least one legitimate state; and
4. A path in the global state graph connects any two legitimate states.

Definition 1.3. A system is *self-stabilizing* iff

1. At least one privilege is present; and
2. Starting from any state, the system is guaranteed to reach a legitimate state after a finite number of moves (convergence).

Property 1 of the definition of self-stabilization, in essence, extends property 1 of legitimate states to illegitimates states.

1.5 The mountain and valley analogy

Draw graph in topological format to depict a global system graph, with illegitimate states as mountains and legitimates states as the valley.

As noted above, the adversary controls scheduling.

Note: The fault injection interval is length of longest path from an illegitimate state to a legitimate state.

Now we need a candidate set of legitimate states, and for that we need a distributed algorithm. We present one next.

Dijkstra critique: Is property 4 of legitimate states necessary?

Try to initiate a discussion about Dijkstra’s definitions.

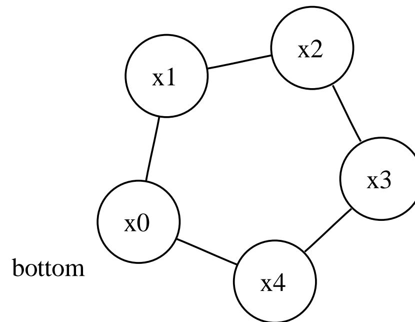

Figure 2: $k > n - 1$ Mutual Exclusion Ring

1.6 Mutual Exclusion Example

In his 1974 paper, Dijkstra provided three algorithms for mutual exclusion, $k > n - 1$, $k = 4$, and $k = 3$, where k is the number of local states and $n > 1$ is the number of systems.

Describe algorithm for $k > n - 1$ depicted in Figure 2 and Algorithm 1 and Algorithm 2. Run through an example, to illustrate legitimate states.

Note that x_i is overloaded in Figure 2: it denotes both a host and the host's state.

Algorithm 1 Ordinary: $x_i, \forall i \in 1..n-1$

if $x_{i-1} \neq x_i$ **then**
  $x_i = x_{i-1}$

Algorithm 2 Extraordinary: x_0

if $x_{n-1} = x_0$ **then**
  $x_0 = x_0 + 1 \mod k$

This is an nonuniform solution: mention uniform vs nonuniform?

TODO 2: *find the uniform vs nonuniform theorem.*

Definition 1.4. For this $k > n - 1$, the legitimate states are those in which only one privilege holds.

1.7 Complexity Metrics

The time complexity of a self-stabilizing algorithm is measured in (asynchronous) rounds or cycles.

Definition 1.5. A *computation step* is internal computation and a single communication operation by a node.

Definition 1.6. An *asynchronous round* in an execution E is the shortest prefix E' of E such that each processor executes at least one step in E'. E'' is the second round of E, and so on.

In other words, a round is a shortest execution trace in which each processor executes at least one step.

Since self-stabilizing systems do not terminate, it is often convenient to measure them in terms of the iterations of their do forever loops.

Definition 1.7. An *asynchronous cycle* in an execution E is the shortest prefix E' of E such that each process executes at least one complete iteration of its do forever loop in E'.

In other words, a cycle is a shortest execution trace in which each processor executes at least one complete iteration of its repeatedly executed list of commands.

1.8 Convergence Proof for $k > n - 1$

Note: With central daemon $k \geq n - 1$ holds.

Lemma 1.1. Every possible configuration c contains a missing state value.

Proof. Inverting the pigeon hole principle, there are only n pigeons to put in the k local state value "slots," so at least one value of k is unused.

□

Lemma 1.2. There must be at least one privilege present.

Proof. Assume that no privilege is present. Then all the ordinary machines, in particular x_{n-1}, must have the same value. But then x_0 is privileged, a contradiction.

□

Lemma 1.3. Starting from any arbitrary configuration with $b \geq 2$ privileges after a finite number of cycles during which the scheduler does not select x_0, either

1. the number of privileges decreases, or
2. machines $n - b$ through n (the b machines with highest index) are privileged (we have a b length privilege "pile-up").

Proof idea: Let p denote a privileged ordinary node, and $\not p$ a non-privileged ordinary node. Then $p\not p \rightarrow \not p p$, while $pp \rightarrow \not p p$ or possibly $\not p \not p$.

Let $b \geq 2$ be the number of privileges in some illegitimate state. b is unchanged by $p\not p \rightarrow \not p p$, but reduced by one or two when pp, a privileged node followed by a privileged node is selected. Thus, the adversary, if it wishes to delay the reduction of privileges, must perform $p\not p \rightarrow \not p p$ until a b length "pile-up" of the privileged ordinary nodes $n - b$ through n.

Lemma 1.4. x_0 cycles through all of its states (all values of k).

Proof Idea:

When Lemma 1.3's "pile-up" occurs, the adversary must either select pp and reduce the number of privileges, or schedule x_0. If the former, the adversary is confronted with the same choice again, until $b = 1$ or he schedules x_0. Note that $b > 0$ by Lemma 1.2. When $b = 1$, the adversary has no choice but to select x_0 infinitely often.

Theorem 1.5. Dijkstra's $k > n - 1$ mutual exclusion algorithm converges.

Proof idea: No matter how many privileges are present, x_0 will cycle through all its states, Lemma 1.4. Thus, x_0 will eventually reach the missing value j, whose existence is established by Lemma 1.1. When $x_0 = x_{n-1} = j$, all ordinary systems in the ring must be also be at state j, since $x_{n-1} = j$ only holds if $x_{n-2} = j$, and so on. This is a legitimate state.

TODO 3: *other properties?*

1.8.1 Counter-Example for $k = 3, n = 5$

Since $k < n - 1$, there is no missing value.

TODO 4: *Rescue my intuition that the cycle length of k must be longer than n's cycle.*

Consider Table 1.

1.9 Impact of Dijkstra's Mutual Exclusion Algorithms

Real world applications months after publication: IBM's token ring was re-implemented to be self-stabilizing. This new implementation resolved several long-standing bugs.

Table 1: Counter-Example for $k = 3, n = 5$

x_0	x_1	x_2	x_3	x_4
$\underline{0}$	0	$\underline{2}$	$\underline{1}$	$\underline{0}$
$\downarrow$				
1	0	2	1	0
				$\downarrow$
1	0	2	1	1
			$\downarrow$	
1	0	2	2	1
		$\downarrow$		
1	0	0	2	1
	$\downarrow$			
1	1	0	2	1
$\downarrow$				
2	1	0	2	1
				$\downarrow$
2	1	0	2	2
			$\downarrow$	
2	1	0	0	2
		$\downarrow$		
2	1	1	0	2
	$\downarrow$			
2	2	1	0	2
$\downarrow$				
0	2	1	0	2
				$\downarrow$
0	2	1	0	0
			$\downarrow$	
0	2	1	1	0
		$\downarrow$		
0	2	2	1	0
	$\downarrow$			
0	0	2	1	0

1.10 Atomicity of Read and Write

So far we have assumed a global scheduler. This is a limiting assumption, ill-suited for a distributed setting.

We can make self-stabilization more realistic and relevant by replacing this assumption with atomic reads and writes.

Algorithm 3 Extraordinary with Atomic R/W

1: nil
2: $\lambda_0 := x_n$
3: nil
4: $\kappa_0 := x_0$
5: nil
6: **if** $\lambda_0 = \kappa_0$ **then**
7: nil
8: $x_0 := \lambda_0 + 1 \mod k$

Algorithm 4 Ordinary with Atomic R/W

1: nil
2: $\lambda_i := x_{i-1}$
3: nil
4: $\kappa_i := x_i$
5: nil
6: **if** $\lambda_i \neq \kappa_i$ **then**
7: nil
8: $x_i := \lambda_i$

How do we resolve the Counter-example in Table 2?

Ring analogy: we split the nodes and repeat the previous proof on a larger ringer; The bound becomes $k > 2n - 1$.

TODO 5: *Could also leave as an exercise for the reader.*

Table 2: Counter-Example for $k = 3, n = 4$ With No Daemon

x_0	x_1	x_2	x_3
e0-6	o0-6	o0-6	o0-6
0	2	1	0
			o7-8
0	2	1	1
		07-8	
0	2	2	1
	07-8		
0	0	2	1
e7-8			
e0-6	o0-6	o0-6	o0-6
1	0	2	1
			o7-8
1	0	1	2
		07-8	
1	0	0	1
	07-8		
1	1	0	1
e7-8			
e0-6	o0-6	o0-6	o0-6
2	1	0	1
			o7-8
2	1	0	0
		o7-8	
2	1	1	0
	o7-8		
2	2	1	0
e7-8			
e0-6	o0-6	o0-6	o0-6
0	2	1	0

1.11 Modern Definitions

A predicate over global system state, defined by the designer of a distributed system, or algorithm, now distinguishes legitimate and illegitimate states.

Definition 1.8. A system is *self-stabilizing* iff

1. *Convergence*: Starting from any state, the system reaches a legitimate state in finite time.
2. *Closure*: From a legitimate state, the system remains in a legitimate state, provided that no fault happens.

How has Dijkstra's initial definition changed?

Dijkstra's properties 1,3,4 of legitimates states have been dropped. Property 1 of his definition of self-stabilization has also been dropped, and property 2 (closure) of legitimate state promoted to replace it.

1.11.1 Anish Auroa

TODO 6: *Suffix-, or prefix-closed? The question is which way does time flow along an execution trace string. The definition of convergence and closed implies left to right, but then we should care about prefix-closed strings, not suffix-closed. Think about this more.*

Definition 1.9. A set of strings L is *suffix-closed*, if $\forall s \in L$, every suffix of s is also in L.

A system S defines a set of (global) states. Σ is a suffix-closed set of strings formed using S as an alphabet. It represents the computations (execution traces) of S.

Definition 1.10. The predicate P is *closed* in S if, $\forall s_i \in S$ and $\forall s_0 s_1 \cdots s_{n-1} \in \Sigma$, $P(s_0) \implies P(s_j), \forall j > 0$.

Definition 1.11. Let P and Q be state predicates of a system S. S is *Q-stabilizing to P* iff

1. *Closure*: P and Q are both closed in S.
2. *Convergence*: Every computation of S that starts at a state in Q has a finite prefix of states not in P.

Q defines a subset of the legitimate states as valid initial states. When $Q =$ true, we essentially get definition 1.8 above.

1.12 Error Detection

The first self-stabilizing algorithms did not detect errors explicitly in order to subsequently repair them. Instead, they constantly pushed the system towards a legitimate state. Since traditional methods for detecting an error were often very difficult and time-consuming [**?**], such a behaviour was considered desirable.

Under the names of local detection [**?**] and local checking [**?**], research proposed new methods for light-weight, explicit error detection for self-stabilizing systems.

In terms of Dijkstra's initial formulation, Varghese' local detection eliminates of Dijkstra's constraint 1 on legitimate states, as well as property 1 of his definition of self-stabilization. Now, the system can be quiescent. His work further generalizes/abstracts Dijkstra's.

By local, we mean that a node need only communicate with its neighbors to detect an error. Local detection methods simplified the task of designing self-stabilizing algorithms considerably, because they separate the design of the error detection mechanism from the recovery mechanism. The new algorithms using local detection also turned out to be much more efficient.

1.13 Global Reset

Related result that any distributed algorithm can be made self-stabilizing using detection and global reset... **TODO 7:** *Find Reference.*

1.14 Composition

TODO 8: *Write-up using Dolev.*

1.15 detectors and correctors

1.16 Conclusion

Other classes of algorithms/protocols for which self-stabilization has been proved:

- Spanning tree
- Leader elections
- Time-adaptive protocols: when only a few errors occur, recovery time should (and can) be made short [5]. The original algorithms of Dijkstra do not have this property.

TODO 9: *Expand this list; use Dolev.*

References

[1] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[2] Wikipedia. Self-stabilization, January 2007. The date records time at which this bib entry was made.