Term Paper:

SELF STABILIZATION IN DISTRIBUTED SYSTEMS

Course Name: Networks & Distributed Computing Systems
Course Number: 17604, Spring 2003
1. Introduction

The concept of “self stabilization” was first introduced by Dijkstra. He defined a system to be self-stabilizing if “regardless of the initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps.” Points to be noted here are that in a distributed system, there is no concept of any global memory. The actions undertaken by a node is based solely on local information available to that node, and these actions must achieve a global objective.

Dijkstra’s original paper had received little or no attention at the time of its publication, and he himself did not elaborate on the usefulness of the concept of self stabilization. However, in the last two decades, this notion has generated a huge amount of interest, and is being used as a formal and unified approach to fault tolerance in distributed systems.

1.1 Definition of self-stabilization

A distributed system consists of several machines, each having its own processor and local memory. There might be some shared memory as well. These processes and their interconnection form a graph which is known as topology of the system. Each machine in the system has a local state. The global state of the system is the union of the local states of its components. The behavior of the system consists of a set of states, a transition relation between them, and a fairness criterion on the transition relation.

Self-stabilization is defined in terms of a system S with respect to a predicate P over its set of global states (where P is supposed to denote correct execution). In order to be self stabilizing with respect to the predicate P, the system S must satisfy the following two properties:

1. **Closure**: P is closed under the execution of S. Once P has been established, it cannot be violated.
2. **Convergence**: Starting from any arbitrary global state, S will reach a state satisfying P within a finite number of state transitions.

(states satisfying P are said to be safe/legitimate, while those that do not, are termed as unsafe/illegitimate).

Stabilization is a generalization of the concept of self-stabilization. Here we have two predicates Q and P, where Q denotes a restricted start condition. Q \( \Rightarrow \) P (read Q stabilizes to P) if

1. **Closure**: P is closed under the execution of S. Once P has been established, it cannot be violated.
2. **Convergence**: Starting from any global state satisfying Q, the system reaches a state satisfying P within a finite number of state transitions.

In a self-stabilizing system with predicate P, TRUE \( \Rightarrow \) P.
Definitions of a few more terms are given below.

**Reachable set:** For a program which is not written with self-stabilization specifically in mind, the definitions of P and Q might not be very clear. The system states which are reachable under normal program execution is termed the reachable set. By definition, the reachable set is closed under program execution.

**Transient failure:** A failure which changes the state of a system, but *not its behavior*. Transient failures can be caused by corruption of local state, or the disruption of message passing channels or shared memory, etc. Self stabilizing systems are designed to recover from transient failures.

1.2 Design Decisions for Self-Stabilizing Systems

Self stabilization is characterized in terms of a “malicious adversary” whose objective is to disrupt the normal operation of the system. [1] This adversary may destroy some portions of the system, or disrupt their operation. Furthermore, it might not be possible for a system to detect that it has been “attacked”, as soon as the attack appears. To be called self-stabilizing, a system must have the capability to recover normal operation in the face of such attacks. If the system (or parts of it) is destroyed completely, so that it is no longer possible for the system to operate, then no self-stabilizing system can work. The adversary wins. However, if enough components are left for the system to operate, then a self-stabilizing system will slowly resume normal operation after the attack. It is up to the designer to decide under what conditions the system may be termed “completely destroyed” or “still capable of operating”.

![Diagram of self-stabilizing system with predicate P](image1)

![Diagram of stabilizing system for Q -> P](image2)
1.3 Why self-stabilization?

Self stabilization has been traditionally used as an approach to handle the following problems. [2]

- **Inconsistent Initialization**: Different processes may be initialized in a fashion inconsistent with each other, or with garbage. Self-stabilizing systems are used to eliminate the overhead of initialization. No matter what the initial state, a self-stabilizing system reaches a legitimate state in a finite number of operations.
- **Mode Change**: A system may be designed to operate in more than one mode. While switching modes (changing functionality), it will be impossible for all processes to simultaneously switch modes. A self-stabilizing system guarantees completion of change of mode in a finite number of operations.
- **Transmission Error**: Errors in transmission leading to corruption of data may be handled by self-stabilizing systems as well.
- **Process Failure and Recovery**: Once a process recovers after failing, its state may be inconsistent with the rest of the system. Self-stabilization guarantees consistency of the process after a finite number of operations.
- **Memory Crash**: If a local memory of a process crashes, its state may become inconsistent with the system. Self-stabilization can be used to recover from such inconsistent global scenarios.

1.4 Self stabilization as an approach to fault tolerance

Self-stabilization provides a generalized non-masking approach to fault tolerance. [1] Non-masking fault tolerance implies that the user feels the effect of transient faults as opposed to paradigms like replication where the effect of the fault is shielded from the user.

Self stabilizing systems have the two following properties:

a) They need not be initialized, and
b) They can recover from transient failures.

Self stabilization represents a departure from the traditional approach to fault tolerance. It does not try to handle each every individual failure separately. Rather, it tries to capture the commonality of all failure modes. Transient failures, whatever their cause, shift the system to an inconsistent state. In the face of transient failures resulting in an inconsistent system state, a self-stabilizing system will eventually correct itself without any form of outside intervention. For example, we can look at the loss of coordination in a distributed system – which is a state where the machines are no longer coordinated with each other. A self stabilizing system can recover from this loss of coordination.
1.5 Self stabilizing communication protocols

To be called self stabilizing, a communication protocol needs to satisfy the following properties.

1. It must be non terminating.
2. There are an infinite number of safe states.
3. There are timeout actions in a non-empty subset of processes.

Most communication protocols, like sliding window protocol, asynchronous handshake are not self-stabilizing. However, self stabilizing versions of these algorithms exist. The alternating bit protocol has a finite number of states, and hence cannot be self stabilizing. A probabilistically self stabilizing alternating bit protocol exists.

1.6 Birth of self stabilization: The original introduction by Dijkstra

Dijkstra’s system consists of a connected graph of processes, each having a finite number of states. Processes directly connected to each other are called neighbors. A privilege is a Boolean function indicating whether a particular transition is enabled for a process. Legitimate states in this system meet a global correctness criterion. There are four constraints on the system.

- In any legitimate state, one or more privileges will be present. This is basically the progress requirement – the system should never terminate.
- In each legitimate state, each possible move will bring the system again into a legitimate state. This is basically the closure definition.
- Each privilege must be present in at least one legitimate state. This corresponds to a minimal number of privileges.
- Each legitimate state is reachable from any other legitimate state. This seems to be a restriction. For example, legitimate states cannot be divided into mutually disjoint subsets.

1.6.1 Token Ring System

Dijkstra used a token ring network with a single token. He noticed that if all processes were identical, then, in general, the problem could not be solved. To counter this, he made one process “distinguished”. Dijkstra’s system consists of $N+1$ processes, each having $K$ states ($K > N$). The processes are denoted by $M(0), … , M(N)$. $M(0)$ is distinguished, and all other processes are identical.

For all $j$, $1\leq j \leq N$, $M(i) \leftarrow M(i-1)$, if $M(i) \neq M(i-1)$. Here, $M(i)$ denotes the state of process $M(i)$.

$M(0) \leftarrow (M(N) + 1) \mod K$ if $M(0) = M(N-1)$. 

It is easy to see that if the system starts with all processes in same state, privilege (token) will rotate around the ring. Also, the system will stabilize if started in an arbitrary state.

Dijkstra’s system has a few drawbacks. Firstly, it requires central control, since transitions of neighboring processes are interfering. By firing its enabled transition, a process can disable an enabled transition in another process. The obvious solution is to have a system whose state transitions are non-interfering. Here, if a state transition is enabled at one instant, it will continue to be enabled until it is executed. Some non-interfering self-stabilizing systems have been described in the literature. [4, 5]

1.6.2 Reducing the number of states/machine in a token ring

It can be shown that if self-stabilization is not a requirement, then there exists an asymmetric token ring with two states per machine. In a self-stabilizing token ring with a central daemon and deterministic execution, it can be shown that a minimum of three states per machine are required. However, for a non-ring topology, the number of states can be reduced to two per machine. There exists a non-trivial self stabilizing system with two states per machine. It requires a high degree of atomicity in each action. Each process reads from three of its neighbors. Thus, obviously, the topology is non-ring.

For a “probabilistically” self-stabilizing synchronous token ring with randomized actions, a solution requiring two states per machine exists. The system is unidirectional and symmetric. [6]

Thus, it can be seen that to obtain self-stabilizing systems with two states per machine, we must either relax the objective to “probabilistic” self stabilization using randomized actions, or use a non-ring topology with higher atomicity in the actions.

1.7 Probabilistic self stabilization

A system S is said to be probabilistically self stabilizing with respect to a predicate P if
1. **Closure**: Same as the definition of closure given earlier.
2. **Convergence**: There exists a function $f$ from natural numbers to $[0,1]$ satisfying $\lim_{k \to \infty} f(k) = 0$, such that the probability of reaching a state satisfying $P$, starting from an arbitrary global state within $k$ state transitions, is $1 - f(k)$.

Algorithms have been proposed for probabilistic orientation of an asynchronous bidirectional ring, as well as for a synchronous ring with odd number of processes and one token.
1.8 Costs of self stabilization

The definition of self stabilization does not put any upper bound on the number of transitions required by the system to reach a safe state starting from an unsafe one. Thus, the system might remain in an unsafe state for a considerable while before reaching a safe state. There are some definitions related to this in the literature.[1, 7]

**Convergence span:** Maximum number of transitions that can be executed in a system, starting from an arbitrary state, to reach a safe state.

**Response span:** Maximum number of transitions that can be executed in a system to reach a specified target state, starting from some initial state.

Thus, the aim of the designer of a self stabilizing algorithm will be to reduce the convergence span and response span.

**Stabilization time**: The worst case time to reach a legitimate state from an arbitrary initial state.

**Stabilization space:** Maximum space used by any legitimate state.

1.9 Alternative Definitions of Self-Stabilization

As an alternative to defining legitimate states using predicates, one may define legitimacy in terms of state behavior, in much the same way as one defines faults.[8] However, such definitions are not easily translatable to local detection of faults.

Further, some additional constraints might be put on a self stabilizing system.[9] It may be required to satisfy:

- **No deadlock:** All executions are infinite.
- **Fairness:** Each legitimate state appears an infinity of times in each computation.

---

1 The idea of time in asynchronous systems is defined here with respect to the number of rounds required for stabilization. The first round is finished when every processor finishes at least one cycle, and round i + 1 finishes when one cycle is finished after round i finishes. A cycle is the minimal sequence of computation that a processor performs for a complete iteration through the specific protocol.
2. Self Stabilization: Modus Operandi

2.1 Methodologies for designing self stabilizing systems

The most commonly used technique used for building self stabilizing programs is layering and modularization. A self stabilizing “platform” is first built. Any program written on that platform automatically becomes self stabilizing.\footnote{[1]}

The self stabilization relation is transitive, i.e. if $P \leadsto Q$ and $Q \leadsto R$, then $P \leadsto R$. Thus, different layers of self stabilizing programs (each by itself self stabilizing) can be composed. The basic idea behind a self stabilizing platform is to provide primitives using which other programs can be written.

There are two basic structuring mechanisms: \textbf{common clock}, and \textbf{topology of interconnection}.

2.1.1 Common Clock

\textbf{Unison} is the process of maintaining time through the use of local clocks for shared memory systems. The properties required here are the \textbf{safety property} and the \textbf{progress property}. For a synchronous shared memory system, these are ensured as follows:

- \textit{Safety}: All clocks have the same value.
- \textit{Progress}: At each step, each clock is incremented by the same amount.

For asynchronous systems with shared memory,

- \textit{Safety}: Clocks of two neighboring nodes can differ by at most 1.
- \textit{Progress}: A clock is incremented to $i+1$ when all neighboring clocks have the value $i$ or $i+1$.

2.2.2 Topology based primitives

\textbf{Leader election} is the basic primitive with respect to an arbitrary dynamic topology. Once a leader has been found, a \textbf{spanning tree} might be computed. Mutual exclusion and reset algorithms can be easily built on top of self stabilizing spanning tree algorithms for arbitrarily connected graphs.

Example: A two layered self stabilizing algorithm for mutual exclusion. The first layer creates a spanning tree from an arbitrarily connected graph, whose topology might change dynamically with the exception of a distinguished process (the root). A breadth first search tree is rooted at the distinguished node. The second layer achieves mutual exclusion on a dynamic tree structured system. It is a token passing based system. The token traverses the tree in depth first manner.
Example: Self stabilizing reset algorithm for asynchronous shared memory system. This has a dynamic topology. It can change, as long as the underlying graph remains connected. There is no distinguished process. The algorithm consists of three layers. The first layer elects the root and forms a spanning tree. The second and third layer, the root passes the reset request on to every node, on detecting any anomaly in the global state.

A self stabilizing “platform” resets the system upon encountering an illegitimate state. Platform writes to variables of original program only if an illegitimate state is detected. Platform does not affect the original program under normal execution.

2.2 A Generic Look at Self-Stabilizing Systems

As specified before, a self-stabilizing system \( P_i \) is self stabilizing with respect to a global predicate \( L_f \) if, starting from any initial configuration, it reaches a closed set of states all satisfying \( L_f \) in a finite number of steps. This problem can be reduced to finding a closed set of states satisfying \( L_f \) in an offline fashion, and dynamically finding the shortest path from any given state to any member of the set. The major flaw with the above scheme lies in the fact that \( L_f \) is usually globally computable, but in most distributed systems not computable locally because the global state is not available.

There are three parts to a self-stabilizing algorithm: detection, convergence and closure. They need not be distinct and may sometimes be difficult to express in terms of local predicates which makes self stabilization a tough problem.

In general, distributed algorithms work based on a set of guard predicates \( G_i(n) \) for every node \( n \). Corresponding to every guard predicate \( G_i \) there exists an action \( A_i \) which is executable locally. In general \( A \) falsifies \( G \). All \( G_i \)s are locally computable. A self stabilizing system based on this framework has to satisfy the following property:

\[
\forall (gs = < ls_1, ls_2, ..., ls_n>) \{ (\sim L_f(gs)) \rightarrow \exists (i, j) \{ G_j(ls_i) \} \}
\]

Alternatively,

\[
\forall (gs = < ls_1, ls_2, ..., ls_n>) \{ \forall (i, j) \{ (\sim G_i(ls_i)) \rightarrow L_f(gs) \} \}
\]

If the system satisfies the above predicate then it has the power of detecting transient faults and vice versa. However, the predicate gives no guarantees about closure or convergence. A condition which guarantees closure is

\[
\forall (gs = < ls_1, ls_2, ..., ls_n>) \{ L_f(gs) \rightarrow \forall (i, j) \{ (\sim G_i(ls_i)) \} \}
\]
Alternatively, 
\[ \forall (gs = < l_{s1}, l_{s2}, \ldots, l_{sn}> \} \{ i, j \} \{ G_i(l_{si}) \} \rightarrow (\sim L_i(gs)) \} \]

However, this is too strong a predicate and is sufficient but not necessary for closure, that too for non-reactive systems. It shows that the system takes no action once the predicate is satisfied. This predicate also does not guarantee convergence. An LTL formula for closure on predicate \( P \) for any continuous sequence of states traversed by the system is:

\[ G (P \rightarrow G P) \]

Convergence cannot be conventionally captured by using first order logic on local predicates. Methods of tackling convergence are best illustrated by looking at proof techniques for self-stabilizing systems. However, an LTL formula for convergence on predicate \( P \) is:

\[ G (\sim P \rightarrow F P) \]

The two formulae together may be looked upon as

\[ (\sim P \cup G P) \]

A system is said to be **pseudostabilizing** or **pseudo self-stabilizing** if the predicate \( P \) eventually holds. This means that \( P \) might hold and not hold a finite unbounded number of times, but eventually it will continue to hold. A pseudo self stabilizing alternating bit protocol exists. For such systems where closure cannot be guaranteed, but eventual convergence is guaranteed, the convergence formula holds and not the one for closure. The formula corresponding to closure becomes weaker:

\[ G (P \rightarrow F G P) \]

The generalized LTL formula for pseudostabilizing systems is:

\[ (F G P) \]

Fig. 2.1: A pseudo self-stabilizing system
2.3 Proof Techniques for Self-Stabilizing Systems

Given a self-stabilizing system, there are a few standard methods to formally prove the property of self-stabilization. These methods are investigated.

The first behavioral property of a self-stabilizing system is that of transient fault detection. Appropriate guard conditions computed locally need to be chosen satisfying the predicate for fault detection. Typically guard conditions are evaluated using only local data and neighbours’ data. For proving convergence and closure, the most common method is that of a monotonic metric.

2.3.1 Monotonic Metric

The most common approach to show that a system is self stabilizing is to carefully choose a metric which is finite, and there exists a bound beyond which the metric only pertains to a legitimate state. [9, 10] It further has to be shown that every action taken by the system leaves the metric unchanged (a finite number of times continuously at most) or makes it progress towards the bound by a finite amount. Then, in a finite number of steps the metric hits the bound and the system stabilizes starting from any initial state. More formally, this may be stated in the following way:

Exhibit a norm function \( f: C \rightarrow W \) where \( W \) is a well-founded set (typically the natural numbers), and \( C \) the set of states of the system such that for each transition \( c_i \rightarrow c_{i+1} \), either \( f(c_i) > f(c_{i+1}) \) or \( c_i \) is a legitimate state.

Let \( < C, ? > \) be a system. If all terminal configurations are legitimate, the legitimacy predicate is closed under \( ? \), and there is such a norm function, then the system is self-stabilizing.

2.3.2 Powerful Local States

If local states may be defined such that global illegitimacy can be captured with a one-to-one relation to local illegitimacy, the problem of convergence and closure is solved. However, such systems are rarely feasible to design.

2.3.3 Proofs for Probabilistic Systems

Proving a system may be probabilistically self-stabilizing corresponds to ensuring that closure holds in a way similar to deterministic systems, but the convergence criteria is modelled by proving that the probability of the system landing up in a legitimate state given sufficient time is 1.
2.4 The role of compilers with respect to self stabilization

The goal of a “self stabilizing compiler” is to convert a non self-stabilizing source program into self stabilizing object code. The idea of a “compiler” is a homomorphism \( f: A \to B \) where \( A \) and \( B \) are two classes of architecture or systems. Then, for each \( m \in A \), \( f(M) \) mimics the actions of \( M \) in some well-defined fashion.

In his seminal work, Dijkstra has shown that there does not exist a compiler from asymmetric rings to symmetric rings that forces or preserves self stabilization. However, if self stabilization is not required, we can compile an asymmetric ring into a symmetric ring.

2.4.1 Compilers for sequential programs

A class of programs exists for which there is a compiler that forces self stabilization while preserving termination. Object programs have runtime and size within a constant factor of source program. We assume here that inputs are incorruptible. These programs must satisfy the following properties.

- Data dependency graphs are acyclic.
- Each rule in the program assigns only one variable.
- For any pair of enabled rules with the same target variable both rules will assign the same value to the variable.

For arbitrary programs, one cannot obtain the same result as for acyclic ones. Consider the class of programs restricted to Boolean variables. It has been shown that if there exists a compiler that forces self stabilization onto Boolean programs, preserving termination, then \( \text{PSPACE} = \text{NP} \), which is not a very likely result. Further, if we require that the source and target to have the same set of variables, then \( \text{PSPACE} = \text{P} \), which is even less likely a result.

However, if we waive the condition of termination, life becomes easier. We introduce the notion of partial fixed points (termination not required). Using this definition one can produce an equivalent program with time complexity and size within a constant factor of the original. See Sections 5.2 and 5.3 for details.

2.4.2 Compilers for asynchronous message passing systems

This compiler should (ideally) compile a non self-stabilizing program into a self stabilizing version in an asynchronous message passing system. \[11\]

Example: A self stabilizing platform, when interleaved with a non self stabilizing program, yields self stabilizing program. The system consists of three components:
• Self stabilizing version of Chandy-Lamport’s global snapshot algorithm. [12]
• Self stabilizing reset algorithm superposed on it. [13]
• The non self-stabilizing program.

The system proceeds as follows: Repeatedly take global snapshots from a distinguished initiator (the assumption is that it is always possible to detect whether a global state is legitimate). If an illegitimate global state is detected, then initiate the reset algorithm. Set the state of the source program to initial state. The compiler requires the program and the predicate (set of safe state) as input.

Extension: Program Q is an extension of program P if the subset of Q corresponding to P behaves exactly like P, except that the same state may repeat. If P terminates, Q needs to repeat the final state of P forever, changing only the variables not in P, in order to achieve self stabilization.

However, it might not be always possible to distinguish between legitimate and illegitimate states. Legitimate states are defined in terms of the reachable states. Unfortunately, computing the reachable set might become intractable. There is another pitfall. Global snapshot captures a future possible set. If the original program stabilizes by itself, we might end up doing a reset from a legitimate state.

2.4.3 Compilers for asynchronous shared memory systems

In this case, we can write the snapshot and reset algorithms in a way very similar to message passing systems. A self stabilizing synchronous shared memory system might be compiled into an asynchronous self stabilizing shared memory system. Assume that processes read and write automatically, and that all shared variables are written to by only one process at a time. The synchronous system is simulated using clocks in unison. One step of the synchronous algorithm is executed each time the clock is incremented.
3. Design Of A Few Demonstrative Self-Stabilizing Algorithms

3.1 A self stabilizing token system [4]

The system described consists of an array of processes. The leftmost process is called bottom, the rightmost one is called top, and all the other processes in between are called middle. The bottom process communicates with only the right neighbor, the top process communicates with only the left neighbor, and the middle processes communicate with both the left and right neighbors. The communication model is synchronous, i.e. for two processes to communicate, both should be ready to communicate with each other. The aim is to implement a mutual exclusion algorithm by means of token passing.

Processes execute the following programs:

\[
\begin{align*}
Bottom &:: \ast [ C ; R ; W ] \\
Middle &:: \ast [ L ; C ; R ; W ] \\
Top &:: \ast [ L ; C ]
\end{align*}
\]

The symbols in the above definition have the following meanings:

* : Infinite looping.
; : Sequential execution
C : Command to execute critical section.
R : Command to communicate with right neighbor. Makes the process wait till the right neighbor reaches the L state. Then both processes simultaneously move to the next command.
L : Command to communicate with left neighbor. Makes the process wait till the left neighbor reaches the R state. Then both processes simultaneously move to the next command.
W : Waiting command. Process waits till right neighbor reaches L. Then it moves to next command, leaving right neighbor in L.

A system state is any string of symbols which satisfies the following regular expression:

\[
(C_b + R_b + W_b)(L_m + C_m + R_m + W_m)^{n-2}(L_t + C_t)
\]

\(n = \text{number of processes}\)

The subscripts \(b, m, t\) denote bottom, middle and top respectively. There exists a special state, called home state which is defined as follows:

\[
C_b L_m^{n-2} L_t
\]

Here, the token resides with the bottom process, and all other processes are in the L state.
The state transitions are defined as follows: (Here \( i \) is either \( b \) or \( m \), and \( j \) is either \( m \) or \( t \))

\( T0: C_i \rightarrow R_i \) \hspace{1cm} (Bottom or middle executes critical section)
\( T1: C_t \rightarrow L_t \) \hspace{1cm} (Top executes critical section)
\( T2: R_iL_j \rightarrow WC_j \) \hspace{1cm} (Adjacent processes communicate).
\( T3: W_bL_j \rightarrow C_bL_j \) \hspace{1cm} (Bottom stops waiting on right neighbor)
\( T4: W_mL_j \rightarrow L_mL_j \) \hspace{1cm} (Middle stops waiting on right neighbor)

A state transition \( x \rightarrow y \) is enabled at the \( k^{th} \) symbol, where \( 1 \leq k \), if there exists \( u, v \) such that \( |u| = k - 1 \) and the state \( s \) is given by \( s = u.x.v \)

\( u.x.v \) is a system state and \( x \rightarrow y \) is enabled. This implies that \( u.y.v \) is a system state. We say, \( u.v.y \) follows \( u.x.v \).

**Reachability:** A system state \( S' \) is reachable from another system state \( S \) iff there is a sequence \( S(0), S(1), \ldots, S(r-1) \), such that \( S(0) = S, S(r-1) = S' \) and \( S(k+1) \) follows \( S(k) \) for all \( k < r-1 \).

Several properties of this system can be proved. These include:

**Theorem 1:** Non interference.
System state \( s. t \) is enabled at symbol number \( k \), Let \( S' \) follow from \( S \), via \( t \), with \( |t| = k - 1 \). \( s. t \) and \( S' \) are unequal, and so are \( k \) and \( k' \). Then \( t \) is enabled at symbol number \( k \) of \( S' \).

**Theorem 2:** Liveness. In every state, at least one transition is enabled.

**Theorem 3:** Mutual exclusion.
Every system state that is reachable from the home state, has at most one \( C \) symbol.

**Theorem 4:** Progress.
For each \( k, 1 < k < n \), starting from the home state, the system will reach a state in which the symbol number \( k \) is \( C \).

**Theorem 5:** Self stabilization
Starting from any state, the system will reach the home state. Thus, the system is self stabilizing. From any unsafe state, the system will eventually reach the home state, which is safe.

For proofs of all these statements, the reader is directed to the original paper.

**3.2 A probabilistically self-stabilizing token system [14]**

A self-stabilizing token circulation algorithm for a ring of identical processes is required to circulate exactly one token in the ring and if in an initial state of ring there are numerous tokens then the algorithm is required to reduce the number of tokens until there is exactly one token.
Problems in a ring of identical processes have been previously considered – e.g. electing a unique leader in a ring of indistinguishable processes. It is well known that for such problems no deterministic solution is possible. Typically probabilistic methods using randomization are proposed as solutions to these problems.

3.2.1 Algorithm

Let $n$ be the number of processes in the ring. It is required that $n$ be odd to obtain an algorithm that is self stabilizing to a single token state. If $n$ is even, then the algorithm self stabilizes to a state without tokens. The state of each process is a single bit and the state of the ring can be modelled by $n$ buts. $x_i$ denotes the element $x_k$ where $k = i \mod n$.

Each process uses a random bit denoted $\gamma_i$ for process $i$. $\Pr(\text{Event})$ is merely the probability of the event occurring.

Let $f$ be the algorithm that takes an input ring state and gives an output ring state. For $0 < i < n$, ring state $y$ satisfies:

$$
\begin{align*}
  y_i &= x_{i-1} & \text{if } x_i \neq x_{i-1} [\text{Cycle the token}] \\
  \gamma_i &= x_i = x_{i-1} [\text{Fair coin toss}] \\

  y &= f.x \text{ gives one an idea about what the next ring state could be given the current state. However, The notation } f^k.x \text{ denotes } k \text{ successive computations of } f, \\
  \text{i.e, } f^k.x &= f^{k-1}.(f.x)
\end{align*}
$$

The algorithm consists of the computation $f^k.x$ where $k$ is unbounded. One possible implementation is the iteration of the (parallel) assignment ‘$x := f.x$’; other methods of computing are also feasible including non-deterministic order in computing the elements of $f^k.x$. We define a token to be any consecutive pair $(x_{i-1}, x_i)$ of equal bits in the ring. Given any ring state, if $n$ is odd then execution of the algorithm results probabilistically in a ring state with one token.

Some notation:

- The ring state is expressed as a vector of bits or a string. [eg. $1^n$ denotes the set of ring states in which all bits are 1]
- Variables $a$ and $b$ stand for complementary bits in an expression. [eg. arguments about the expression $a^p b^q$ apply to either $0^p 1^q$ or $1^p 0^q$.]
- Any rotation of a string specifies the same ring state [eg. apbq and bpqa represent the same set of states]. Le
- $c$ is a placeholder bit. [eg. acc represents aaa, aba, abb, and aab.] The expression $c^0$ represents the set of all ring states. $c^0$ denotes an empty string.
- The notation for a quantified expression is ( op : range : term ) where op specifies the operator and dummy variable for the quantification, range
specifies the range of the dummy, and term is an expression that is some function or predicate dependent on the dummy.

- The operators used are \( S \) for summation and \( N \) for counting the number of occurrences that a predicate term holds in the specified range.
- Given a ring state \( x \), a token is at process \( i \) if \( x_i = x_{i-1} \); and there is shift at process \( i \) if \( x_i \neq x_{i-1} \).
- The number of tokens in the ring is denoted by \( S.x : S.x = (N_i : 0 < i < n : x_i = x_{i-1}) [0 < S.x < n] \)

**Lemma 0** \( S.x \) mod 2 = \( n \) mod 2. If \( n \) is odd the number of tokens are odd, and if \( n \) is even there cannot exist 1 token in the system. Any procedure that changes the number of tokens will change the number by some multiple of two.

Proof: The lemma obviously holds in the case \( S.x = n \). In the case \( S.x < n \), ring state \( x \) is a member of the set of states given by alternating strings of ‘a’ and ‘b’ bits i.e.

\[
a^{M,0}b^{M,1} \ldots a^{M,(k-2)}b^{M,(k-1)}
\]

where \( M \) is a vector of \( k \) positive integers subject to the constraint

\[
(1 < k) \land (R < n) \land (S_j : 0 < j < k : M_j) = n
\]

**Theorem 0 (Safety)** Tokens do not increase in number by computation of \( f \).

\( S.x \geq S.(f.x) \)

Proof: The theorem trivially holds in the case \( S.x = n \). In the case \( S.x < n \) ring state \( x \) belongs to the set of states

\[
a^{M,0}b^{M,1} \ldots a^{M,(k-2)}b^{M,(k-1)}
\]

as described in the proof of Lemma 0. From the definition of \( f \) and substituting \( c \) for each evaluation of a random bit the state \( f.x \) belongs to the set of states

\[
bc^{M,0-1}ac^{M,1-1} \ldots bc^{M,(k-2)-1}ac^{M,(k-1)-1}
\]

The Safety Theorem and Lemma 0 demonstrate that if there is exactly one token in ring state \( x \) then all subsequently computed states \( f.x \) also have exactly one token. The following theorem shows that

**Theorem 1 (Progress):** Computation of \( f \) probabilistically circulates a token in the ring.

\[
S.x = 1 \land x_i = x_{i-1} \Rightarrow \Pr(R : k > 0 : y = f^k x \land y_i = y_{i+1}) = 1
\]
Proof: The theorem holds trivially in the case \( n = 1 \); henceforth suppose \( n > 1 \).
Since the token is at process \( i \) in state \( x \), the probability that the token is at process \( i \) in state \( f.x \) is either \( (1 - r_i) \) or \( (r_i) \) (the complementary outcome of \( f.x \) places the token at process \( i + 1 \)). Therefore it suffices to show
\[
\Pr(\exists j > 0 : y = f^j.x \land y_{i-1} = y_i) = 0
\]
Let \( u \) be the maximum of \( r_i \) and \( (1 - r_i) \). For any \( m > 0 \),
\[
\Pr(\exists 0 < j < m : y = f^j.x \land y_{i-1} = y_i) \leq u^m
\]
Hence the theorem follows from
\[
\lim_{m \to \infty} u^m = 0
\]

The function \( D \) is introduced to partition the set of ring states with at least two tokens. \( D.x \) is defined as the minimum distance between two tokens in ring state \( x \). If state \( x \) has fewer than two tokens then \( D.x \) is undefined. If state \( x \) has a pair of adjacent tokens, then \( D.x = 1 \), that is, if \( x \) has a string expression of the form \( a^n b^m \) then \( D.x = 1 \). \( D.x \) is at most \( n/2 \). The following lemma shows that the distance between tokens decreases probabilistically.

**Lemma 1**
\[
D.x = p \land p > 1 \implies \Pr(\exists k > 0 : D.(f^k.x) < D.x) = 1
\]

**Proof:** By expanding the definition of \( f \) under the condition \( D.x > 1 \), that \( \Pr(D.(f.x) < D.x) > 0 \) (for instance, if process \( i \) and process \( i + k \) have tokens and there are no tokens between \( i \) and \( i + k \), then one possible outcome of computing \( f \) is that process \( i + 1 \) has a token and process \( i + k \) has a token).

By induction on \( p \).

**Basis:** \( p = n/2 \). By a similar argument to that given in the proof of the Progress Theorem,
\[
\Pr(\exists k > 0 : D.(f^k.x) = n/2) = 0
\]

**Induction:** \( 1 < p < n/2 \). Suppose we have some computation that satisfies \( (\exists k : D.(f^k.x) \geq D.x) \). By the inductive hypothesis, for every \( i \) such that \( D.(f^i.x) > D.x \),
\[
\Pr(\exists j > 0 : D.(f^j.(f^i.x)) < D.(f^i.x)) = 1
\]
Consequently, with probability one, there are infinitely many \( m \) such that \( D.(f^m.x) = D.x \). The number of states is finite, so some state \( z \) that satisfies \( D.z = D.x \) is visited infinitely often in the supposed computation. The probability that \( D.(f.z) = D.z \) is non-zero; consequently the probability that state \( z \) is visited infinitely often, without the outcome \( D.(f.z) < D.z \), is zero.

Therefore the probability of the computation we have supposed is zero.
Theorem 2  (Convergence Theorem) $S.x > 1 \Rightarrow \Pr(\exists k : k > 0 : S.(f^k.x) < S.x) = 1$. This shows that with probability one, from any initial state, execution of the algorithm eventually arrives at a state with at most one token.

Proof: Consider some state $z$ satisfying $D.z = 1$. From the proof of the Safety Theorem $D.z = 1$ implies there exists some $i$ such that $M.i > 3$ in a term $bc^{M.i-1}a$ or a term $ac^{M.i-1}b$ a possible outcome of the computation $f.z$ is the assignment of values to the c-bits that increases the number of shifts in the string expression, thereby decreasing the number of tokens. Therefore $\Pr(S.(f.z) < S.z) > 0$. Lemma 1 can be applied repetitively to conclude, with probability one, $D.(f^m.x) = 1$ for some $m$; there is a non-zero probability that $S.(f^{m+1}.x) < S.(f^m.x)$. By a similar argument to that given in the proof of Lemma 1 it follows that

$$S.x > 1 \Rightarrow \Pr(\exists k : k > 0 : S.(f^k.x) = S.x ) = 0$$

3.3 Alternate Bit Protocol [15]

The self stabilizing version of the Alternate Bit Protocol serves as an illustrative example of a self-stabilizing algorithm. Alternate Bit Protocol is a special case of sliding window protocol. In this protocol, an aperiodic sequence of numbers taken from a bounded size alphabet is generated. Here the transmitter is an infinite state machine.

The main purpose of a data-link protocol is to transmit data reliably from one end to another. The term ‘reliably’ is used here in the generic sense which spans from the arrival of the message error free, without any loss and the preservation of the message order.

3.3.1 Underlying Model:

The system consists of two processes: the transmitter(T), the receiver(R) two types of links, T-R and R-T oriented in the reverse directions of each other.

During the transmission process the message can be lost but the messages that are not lost are delivered properly. In this protocol, the transmitter is associated with a tape of an infinite sequence of data elements $I=\{D_0,D_1,\ldots\}$. The transmitter reads the data sequence and sends the data one by one and the receiver writes the data in the output tape $O$ of receiver. The main goals of a data-link protocol are:

- Safety: The output tape sequence of receiver $O$ is always prefix of that of $I$. 

19
• Liveness: For proper fairness condition, every data element $D_i$ is written into $O$.

Formal Protocol:

The generic data-link protocol consists of two processes: a transmitter $T$ and a receiver $R$ each code consisting of a set of actions. Each action can be enabled or disabled in accordance with the state of the system. An enabled action may start its execution provided that the actions of the system are executed without any intervention, i.e. atomic. Each action has the form guard $\rightarrow$ action. The guard may be a Boolean variable or a receive statement. The guard timeout is provided for the timeout option. The commands consist of: assignment statement, send primitive of the form send(-), input primitive of the form read(-), output primitive write(-), and the function choose(?/x) which chooses a sequence no. other than $x$ from the sequence alphabet ?. The command reset_timeout reinitializes the timeout clock and the timeout variable.

Transmitter $T$:

Seq int init 0;
 i int;
 msg data;
read_next(msg);

timeout $\rightarrow$ send(seq, msg);
 reset_timeout;
rcv(i) $\rightarrow$ if (seq = i) then
 seq = choose(?/seq);
 read_next(msg);
 send(seq, msg);
 fi
 reset_timeout;

Receiver $R$:

ack int init 1;
 j int;
 rmsg data;

rcv(j) $\rightarrow$ if (ack ? j) then
 ack = j;
 write_next(rmsg);
 fi
 send(ack);
The self stabilization property of the above protocol can be demonstrated provided the sequence number generated by the choose function is aperiodic. A few notations:

- The state of the transmitter is represented by the sequence number seq.
- The state of the receiver is represented by the acknowledgement ack.
- The state of the T-R link is represented by the sequence \( a_1, a_2, \ldots, a_n \), \( a_i \neq \).
- The state of the R-T link is represented by the sequence \( \beta_1, \beta_2, \ldots, \beta_m \), \( \beta_i \neq \).

The state \( S \) of the system is the combination of the states of the processes and the links:

\[
S = \text{seq} | a_1 a_2 \ldots a_n | \text{ack} | \beta_1 \beta_2 \ldots \beta_m
\]

The allowable actions are represented by a set of transition relations: \( d: S \times S \). Only six possible transition relation exists as given below:

- \( d_1 \): the transmitter transmits a message.
- \( d_2 \): the transmitter receives an ack equal to the seq. it sends the next message in sequence.
- \( d_3 \): the transmitter receives an ack that is not equal to seq.
- \( d_4 \): the receiver receives the message and sends an ack.
- \( d_5 (i) \): the \( i \)-th message is lost in T-R link.
- \( d_6 (j) \): the \( j \)-th message is lost in R-T link.

A schedule \( s \) is a finite or infinite sequence of transitions. The configuration resulting from state \( S \) with the application of \( s \) is denoted by \( S_s \). In the state, where the messages are identical the system state \( S \) can be replaced with a compact representation \( CS(S) \) by replacing the maximal sequence of identical seq number with only one instance of the seq. number. The transition \( d_i \) that causes a change in the compact state is denoted by \( d'_i \).

The Rank function is defined as:

\[
\text{Rank}(S) = \begin{cases} 
I(S) + 1; & \text{if first}(S) = \text{last}(S) \\
I(S); & \text{otherwise}
\end{cases}
\]

Fairness condition on the transmitter and the receiver: if an action on the transmitter or the receiver is enabled continuously, it will be eventually executed. Fairness condition on the communication link: if a message is sent out continuously through a communication channel, it will eventually reach the targeted node.

**Theorem**: In a state \( S \) such that \( \text{Rank}(S) = 2 \), the protocol satisfies its liveness property. Furthermore, the system is guaranteed to reach a legal state from which on the protocol satisfies its safety requirements.

**Proof**: There are three types of \( CS(S) \) with \( \text{Rank}(S) = 2 \). They are:
• CS(S) = αβ; ack = β : by fairness condition, the repeated sending of α, it will be received eventually. The system will reach the state S' where CS(S') = α. This implies that the receiver will write α to O.
• CS(S) = αβ; ack = α : Here also the system will reach S': CS(S') = α finally.
• CS(S) = α : by the fairness assumption on the link we can say the massage sent will eventually reach and the new state is CS(S') = α'α; ack=α.

The system must eventually reach a state S, where CS(S) = α and every message has data msg. The required safety property that the output sequence O on the receiver is a subsequence of input sequence I in transmitter will be held from this point on which shows the self stabilization property.
4. The Route To Self-Stabilization

The major problem with self-stabilizing systems is that they provide only the guarantee of eventually returning to a safe state. No guarantee is given as to when or how – two important performance issues for any algorithm. The ideas of Superstabilization and Fault Containment address just that.

4.1 Fault Containment

Fault Containment addresses the idea of how long it takes for a system to recover from a transient fault. [7] Given the non-masking nature of self-stabilizing fault tolerance, one would ideally want levels of disruption proportional to the transient fault. Self-stabilization provides no such guarantees, hence the need for fault containment. The effect of transient faults is tightly contained by fault-containment – which may be either in terms of time needed to recover from the fault or number of processes affected.

4.1.1 Definitions

The local state of a self-stabilizing system <p, s> is partitioned into primary and secondary portions: The primary portion pertains to those variables required for computation and output of the distributed system. The secondary portion pertains to those variables required for self-stabilization.

A k-faulty state of a system is any state which may be obtained by corrupting the local state of k processes when the system was in a legitimate state.

If \( P_f \) be the self-stabilizing protocol, and \( L_f \) be the predicate defining the legitimate states, a predicate \( L \) may be declared such that:

- \( \forall (p,s) \ L_f(p,s) \rightarrow L(p,s) \)
- \( L \) depends only on the primary portion, i.e. \( \forall (p,s,s') \ L(p,s) == L(p,s') \)
- \( L \) is closed with respect to \( P_f \)

The protocol \( P_f \) is fault containing with respect to \( L \) if from any 1-faulty state of \( P_f \), the system reaches a state satisfying \( L \) in at most \( O(1) \) time.

Containment time is the worst case time for a self stabilizing system to arrive at a state satisfying \( L \) from any 1-faulty state. This is a measure of how quickly an observable fault is corrected by \( P_f \) and is \( O(1) \) for fault-containing systems.

Contamination number is the worst case number of processes that change the primary portion of their state during recovery from a 1-faulty process to a state satisfying \( L \). This is a measure of how many processes make observable changes to their states before a 1-faulty state is repaired.
Fault gap is the worst case time for a 1-faulty state to reach a legitimate state (satisfying $L_1$). Fault gap $\geq$ Containment Time. It is a measure of how minimally far apart in time two 1-process faults may occur for fault containment to take place.

Containment Space is the maximum size of the secondary portion of the local state of a process for all legitimate states – space overhead for fault containment.

4.1.2 Generalized Fault Containment

The idea of generalized fault containment is to construct an algorithm such that given any self-stabilizing system $P$ with predicate $L'$ as input, it will give a self-stabilizing and fault-containing version of the system $P_f$. The global state space of $P$ is the primary state space of $P_f$. Also, $\forall(p,s) \ (L(p,s) = L'(p))$.

$P_f$ may be viewed as two protocols $C$ and $C'$ executing one after the other. $C$ is a synchronous, non self-stabilizing protocol that, when started in a 1-faulty state of $P$ takes it to a state satisfying $L$ in constant time. Then $C'$ takes the system to a legitimate state. If the system is started in a state which is not 1-faulty, trivially $C$ is not executed and only $C'$ is executed. $C$ and $C'$ are synchronized using a stabilizing counter.

Every process has a counter $t_i$ which is said to be consistent if it differs by at most one from its’ neighbours’ counters. The goal of the counter is to synchronously raise the counter value to a positive integer $M$, and then decrement it synchronously (in terms of rounds) to zero.

The algorithm is as follows:

Set $t_i$ to $M$ when any of the following happen:

- If some neighbour $j$ of $i$ has $t_j = M$ and $i$ has $t_i < M - n$, i.e. inconsistent
- If $t_i$ is greater than and inconsistent with a neighbor’s $t_j$ and $t_i < M - n$, i.e. inconsistent
- If $t_i$ is zero and (either a local guard condition for protocol $P$ is enabled or state is not consistent with protocol $C$), i.e. some fault has corrupted the state variables

Decrement $t_i$ and take corresponding action($t_i$) when the following happens:
• If $t_i$ is consistent and greater than 0 and $\geq$ all neighboring $t_j$s i.e. synchronously execute the countdown of the timer and corresponding actions
• If $t_i \geq$ all neighboring $t_j$s and all neighboring $t_j$s $\geq M - n$, i.e. even though inconsistent, decrements and prevents livelock by not allowing a $t_i$ value $M$ to circulate forever in a cycle.

For different ranges of $t_i$, the action($t_i$) taken is as follows:

$[M - c + 1 .. M]$ $c$ is the worst case running time of $c$ in synchronous rounds. In this interval, at time $t_{m,M}$ step $m$ of $C$ is executed.

$[b+1 .. M - \max(c,n)]$ Since $C$ is definitely over by this time, the self-stabilizing algorithm may start. Action A is executed in this range provided guard condition $G_i$ holds. $M$ has to be sufficiently large for the protocol $P$ to complete operation inside the range. If $n > c$, extra rounds are simply idled away.

$[2 .. b]$ Book-keeping operations to legitimize the variables of $C$ are undertaken. This could have been undertaken in the first range, but was not because the actions in the second range may have been conflicting. $b-1$ is the maximum number of bookkeeping operations required.

$[1 .. 1]$ No action, system has converged.

$C$ also has to be designed in constant time. We may design a 3 round $C$ as follows. Club all local variables into a single variable $x$. Use a local vector $s$ to keep a copy of each neighboring $x$. In round 1, a process $i$ with more than one neighbor can correct $x_i$. Faulty $s$ variables are corrected in round 2, and processes with one neighbor correct their $x_i$ in round 3.

4.1.3 Tradeoffs

In the above algorithm, containment time is $O(1)$, contamination number 1, containment space $O(d.x)$ [where $d$ is maximum degree and $x$ is maximum space used by $P$] which may be improved to $O(1)$, and fault-gap $O(T.D)$ [ $D$ = diameter of graph ]

However, fault containment comes for a price. The following impossibility results are well known:

• There exists no fault-containing version of a self-stabilizing algorithm (with stabilization time $T$) such that the fault gap is $O(1)$ and stabilization time $T + constant$. 

There exists no fault-containing version of a self-stabilizing algorithm (with stabilization time $T$) such that the contamination number is $O(1)$ and stabilization time $T + \text{constant}$.

4.2 Superstabilization

Superstabilization addresses the issue of how a system recovers from transient faults. The basic idea is that since self-stabilization is a non-masking approach to fault tolerance, there is no guarantee on the “goodness” of the system on the route to stabilization. Superstabilization aims to give such a guarantee. [16]

4.2.1 Definitions

A trajectory is a sequence of global states which the system passes through which are piecewise either fair computation or topology change events.

Superstabilization is a stricter form of self-stabilization. It is usually defined in terms of a passage predicate which all global states must satisfy on the path to recovery. A system is superstabilizing with respect to a class of topology changes $T$ iff

- It is self-stabilizing.
- For every trajectory beginning at a legitimate state, and containing a single topology change of type $T$, the passage holds for every state in the trajectory.
  
  If the self-stabilization predicate is $S$, and the passage predicate is $P$, the LTL formula is given by:

$$G ( \neg S \rightarrow G ( P U S ) )$$

A system is continuously superstabilizing with respect to a class of topology changes $T$ iff

- It is self-stabilizing.
- For every trajectory beginning at a legitimate state, and containing only topology changes of type $T$, the passage holds for every state in the trajectory.

Superstabilization time is the maximum amount of time it takes for the protocol to reach a legitimate state starting from a legitimate state followed by a single topology change.
**Adjustment measure** is the maximum number of processors that must change their local states, upon a topology change from a legitimate state, so that the system is in a legitimate state.

### 4.2.2 Issues in Superstabilization

The passage predicate thus is a weaker condition than the self-stabilization predicate, but strong enough to be useful (for eg, if it is true for every element of the state space, it is no use at all). For a token-based system, the passage predicate could be the existence of at most one token in the system, and the stabilization condition that of exactly one token in the system. The passage predicate merely specifies that the system is “nearly legitimate”. The fact that every state in a valid computation has to satisfy the passage predicate while recovering from a topology change, guarantees that the system gets no worse until it gets better.

Secondly, Superstabilization is defined explicitly with respect to topology changes, not any other kind of transient fault. The definition may be easily extended, but making superstabilizing systems with respect to arbitrary transient faults is difficult.

Superstabilization assumes that processes can sense local changes in topology – which is usually a valid assumption. Four kinds of interrupts are possible: \( \text{fail}_p \), \( \text{fail}_{pq} \), \( \text{recover}_p \) and \( \text{recover}_{pq} \). \( \text{fail}_p \) corresponds to when a process \( p \) goes down, and \( \text{recover}_p \) when it recovers. \( \text{fail}_{pq} \) corresponds to when the link between processes \( p \) and \( q \) go down, and \( \text{recover}_{pq} \) when it recovers. \( \text{fail}_p \) causes all interrupts of the kind \( \text{fail}_{ab} \) where either \( a \) or \( b \) is \( p \). Similarly, \( \text{recover}_p \) causes all interrupts of the kind \( \text{recover}_{ab} \) where either \( a \) or \( b \) is \( p \). \( p \) is said to be incident on all interrupts due to topology changes local to \( p \).

### 4.2.4 A superstabilizing algorithm

Let \( \mathcal{C} \) be the upper-bound on the number of neighbors a node may have. Let \( \mathcal{C} \) be an ordered set of colors such that \( | \mathcal{C} | \geq \mathcal{C} + 1 \). Every processor \( p \) has a register \( \text{color}_p \) which it can write to and its neighbors \( N_p \) can read. The self-stabilizing coloring scheme is defined with respect to the predicate \( \forall p,q \in \mathcal{P} \{ (q \in N_p) \rightarrow \text{PRED}(p,q) \} \) where:

\[
\text{PRED}(p,q) \equiv (\text{color}_p \neq \text{color}_q) \land (\text{color}_p \in \mathcal{C}) \land (\text{color}_q \in \mathcal{C})
\]

and further, any computation starting with a state satisfying the above predicate should not further change color.
However, a single topology change may upset the predicate. So, we augment values taken by a color variable by $\mathcal{C} \cup \{ $ \}, where $ represents “no color presently”. The Superstabilization predicate may thus be given as

$$\forall p,q \in N_p \rightarrow \text{PRED}'(p,q)$$

\[
\text{PRED}'(p,q) \equiv \text{PRED}(p,q) \lor \text{color}_p = $ \lor \text{color}_q = $
\]

This is a weaker predicate than the self-stabilizing predicate and is typically defined by disjunctions with the self-stabilizing predicate.

The algorithm is as follows:

**Process**

A, B $\leftarrow$ F

For each q $\in$ N_p

\[
\{
\begin{array}{l}
\text{Read color}_q \\
A = A \cup \{ \text{color}_q \} \\
\text{if (q} > \text{p)} B = B \cup \{ \text{color}_q \}
\end{array}
\}
\]

if NOT ( color)_p $\in \mathcal{C} $

Choose color_p from $\mathcal{C} - A$

if ( color)_p $\in B$

Choose color_p from $\mathcal{C} - A$

Set color_p

Interrupt section:

if ( interrupt == recover_p)

\[
\text{color}_p \leftarrow \text{some element of } \mathcal{C}
\]

if ( interrupt == recover_{pq}) $\land (p > q)$

\[
\text{color}_p \leftarrow $
\]
4.2.5 Generic Superstabilization

There are ways of achieving superstabilizing algorithms from any self-stabilizing algorithm $P$. The idea is to use a superstabilizing topology broadcast on top of the self-stabilizing algorithm. Briefly, the algorithm proceeds in 3 phases:

**Phase 1:** This corresponds to the normal state of the superstabilizer where $P$ is active.

**Phase 2:** A freezing protocol is initiated on topology change and collects snapshots from the frozen processors. A leader among processes incident on a topology change is chosen as coordinator for the next phase.

**Phase 3:** Computes a new global state for $P$, and distributes the corresponding local states to the processes. Once this is complete, the protocol reverts back to phase zero.
5. Self Stabilizing Computation: A Theoretical Approach

5.1 Limitations Of Self-Stabilizing computation

5.1.1 Lack of fault-containing self stabilization for k-faulty systems

The most common drawback of self-stabilizing systems is that it provides no bounds on the time after which it will recover from a transient fault or an illegal initial configuration. The commonest approach to deal with this is fault containment, as elucidated before. In general, for real-time systems which may be k-faulty (k>1), designing fault containing self stabilizing algorithms which have small convergence time and containment time may not always be possible.

5.1.2 Role of Asymmetry

The role of asymmetry in self stabilization is an important issue. Dijkstra demonstrated that a self stabilizing token ring with non-prime number of processes must have an asymmetric design (that is all processes cannot be identical). Burns and Pachl demonstrated that there is a self stabilizing ring with no distinguished process (a symmetric solution) if the size of the ring is prime. [6, 17] Thus it appears that in general, asymmetry is required for achieving self-stabilization. Moreover, it can be shown that in deterministic systems, asymmetry must be maintained in systems where processes synchronize with each other (e.g. mutex problems like dining philosopher’s problem). There can be two methods for achieving asymmetry:

1. **Asymmetry by state**: All machines are identical, but have different initial local states. In general, these systems cannot be self-stabilizing, for if they are put into a global state with symmetric local states, they will not self-stabilize.

2. **Asymmetry by identity**: Not all machines are identical. Identical programs can be parameterized by a local id. Depending on their own id and other observed ids, different processes may act differently. These systems can be self stabilizing.

Asymmetry is required because the legitimate global states may be asymmetric in its local states, and therefore, starting from symmetric initial conditions, running symmetric algorithms, only global states with symmetric local states may be reached. [18] A symmetric system may thus be trapped in only symmetric states if it starts from symmetric initial local states. Asymmetry is not necessary in probabilistic systems, for the probabilistic behavior removes symmetry from the system in the long run.

5.1.3 Control Theory Perspective

In control theory perspective, a distributed system may be looked on as
a dynamic system – with its successive state changes being mapped with respect to time. A self-stabilizing system thus inevitably has as its sole attractor the set of legitimate states (i.e. any computation in the long run reaches a state in the set of legitimate states and stays there). For certain systems (distributed systems operating under a particular algorithm) there may be more than one attractor or fixed point or period – which may inhibit the self stabilizing behavior of a system. This may occur in the following ways:

1. **Deadlock**: There is a fixed point where the system may get stuck. Several processes may wait for messages that never arrive, or resources that are never released. Under these circumstances, it cannot self stabilize.

2. **Livelock**: There is a stable period in the system where the system may be stuck. A set of processes may exchange messages and perform computations in an infinite loop without achieving anything.

3. **Termination**: If the dynamics of the system terminate at an illegitimate state, i.e. it reaches a final state which is unsafe, it cannot self stabilize.

5.1.4 **Isolation**

Cases may occur where the local state of a process is consistent, but the global state is not. The system is unable to stabilize due to inadequate communication or coordination between processes.

5.2 **Automatic Self Stabilization**

The basic goal is to convert any given algorithm to a self-stabilizing one. [11] A self-stabilizing algorithm is more powerful in the sense that it can tolerate transient faults. One easy way to do it is to modify any algorithm P as follows:

Let k be the upper bound on the number of steps executed in the algorithm.
probation $\leftarrow k$
For every step $P_i$ in algorithm P
{
    if current state is unsafe
        probation $\leftarrow$ probation -1
    else
        probation $\leftarrow k$
    if ( probation == 0)
        reset the algorithm
    execute $P_i$
}
However, this approach loses the entire computation if a transient fault occurs. Also, there is no question of communication since this algorithm is centralized. This approach can further be refined as follows.

5.3 Computational Power of Distributed Self-Stabilizing Systems

It may be shown that any Universal Turing Machine run may be simulated in a self-stabilizing fashion in a distributed system by resetting the machine every time a state repeat in a computation is detected. [19] The memory used by each process in the distributed system is constant. The number of processes n may be specified beforehand. If the Universal Turing Machine uses more than n tape cells to compute the given algorithm, or is non-terminating, each process writes ‘-’ as output, otherwise each process generates part of the output in a distributed fashion. Each process also takes part of the input as distributed fashion.

We consider a distributed system of processes P₁, P₂, ..., Pₙ which are connected in a chain. The chain may be part of a ring or a network but it is important that it has two ends and a circulating token to generate asymmetry. It is not necessary for the processors to know the number of other processors, but each processor requires the sense of left and right direction. We will consider P₁ to be the leftmost or leader process, with P₂ to its right and so on upto Pₙ which is the rightmost or tail process.

5.3.1 Definitions

Every process Pᵢ contains an input register Iᵢ and an output register Oᵢ. The input and output registers can store any element of the Turing Machine alphabet set or nothing signified by ‘-’. In addition every process contains 2 bits of a distributed counter Cᵢ, HdMrkᵢ which is a bit signifying whether a process is the head or not, Wᵢ which is the working tape symbol at that node, Movᵢ which indicates the computed movement of the head [LEFT, RIGHT or STAY], Rᵢ which stores the value of the token received, and Transᵢ which is a local copy of the transition table of the Turing Machine.

The token value is a combination of four values: B which is the binary carry for the distributed counter, restart which is a bit signifying whether or not to restart, and state which encodes the current state of the Universal Turing Machine and requires \( \log_2 N \) bits to encode where N is the total number of states in the Turing Machine. state is set to ‘-’ during a restart. The token also carries head movement indication as a Boolean value in HdEnable.

5.3.2 Initialization

\( n \geq l \), where \( w \) is the length of the input word \( w \) [since the Turing Machine does not use more than \( n \) tape cells ], \( w_i \) is the \( i^{th} \) symbol of the input word. We
can modify \( w \) as follows: \( w = w \cdot n - l \). Then \( w \) has exactly \( n \) symbols. Initialization corresponds to the assignments \( l_i \leftarrow w_i \).

5.3.3 Execution

The processors are initialized to start a new simulation. Initialization does not affect the output registers. The result of the simulation is written to the output registers. A single step of the Turing Machine is simulated every time the token travels from the leader to the tail and back. Once every round, it increments the distributed counter. At any point of time, at most one process is enabled as head. That is the position where the head of the virtual Turing Machine resides. The movement of the head is conveyed by the token. Only that process which is the head can change its tape cell content, determine acceptance/ rejection, and specify the direction of head movement and accordingly send the message to its neighbor when it can next.

5.3.4 Algorithm

\[ P_1 : \]

On receipt of token from \( P_2 : \)

\[
\begin{align*}
R & \leftarrow \text{token} \\
\text{if} & \ ( R.B == 1 ) \ \text{AND} \ ( C == 11) \\
& \text{token.restart} = \text{TRUE} \\
\text{if} & \ ( \text{token.state} ? \{ \text{ACCEPT, REJECT} \} ) \\
& \text{token.restart} = \text{TRUE} \\
C & \leftarrow C + 1 \\
\text{if} & \ ( \text{HdMrk} == \text{TRUE}) \\
& \{ \\
& \text{token.state} , W , \text{Mov} \leftarrow \text{Trans [R.state][W]} \\
& \} \\
\text{if} & \ ( \text{Mov} == \text{RIGHT}) \\
& \{ \\
& \text{HdMrk} \leftarrow \text{FALSE} \\
& \text{token.HdEnable} \leftarrow \text{TRUE} \\
& \} \\
\text{if} & \ ( \text{Mov} == \text{LEFT}) \\
& \{ \\
& \text{token.restart} \leftarrow \text{TRUE} \\
& \} \\
\text{if} & \ ( \text{r.HdEnable} == \text{TRUE}) \\
& \{ \\
& \text{HdMark} \leftarrow \text{TRUE} \\
& \text{token.HdEnable} \leftarrow \text{FALSE} \\
& \} \\
\text{if} & \ ( \text{token.restart} == \text{TRUE}) \\
& \{ \\
& W \leftarrow I \\
& C \leftarrow 00 \\
& \text{HdMrk} \leftarrow \text{TRUE} \\
& \text{if} \ ( \text{state} ? \{ \text{ACCEPT, REJECT} \} )
\end{align*}
\]
\{ 
  O \leftarrow \text{Trans}[R.\text{state}]
  \text{token.state} \leftarrow \text{Initial state}
\}

\text{else} \\
\{ 
  O \leftarrow \text{'}\text{'}
  \text{state} \leftarrow \text{'}\text{'}
\}

\text{Send token to P}_2
\}

\text{P}_n : 

\text{On receipt of token from P}_{n-1} : 
\{
  R \leftarrow \text{token}
  \text{if (HdMrk == TRUE)}  \\
  \{ 
    \text{token.state} , W, \text{Mov} \leftarrow \text{Trans} [R.\text{state}][W]
  \}  \\
  \text{if (Mov == LEFT)} \\
  \{ 
    \text{HdMrk} \leftarrow \text{FALSE}
    \text{token.HdEnable} \leftarrow \text{TRUE}
  \}  \\
  \text{if (Mov == RIGHT)} \\
  \{ 
    \text{token.restart} \leftarrow \text{TRUE}
    \text{HdMrk} \leftarrow \text{FALSE}
  \}  \\
  \text{if (r.\text{HdEnable} == TRUE)} \\
  \{ 
    \text{HdMark} \leftarrow \text{TRUE}
    \text{token.HdEnable} \leftarrow \text{FALSE}
  \}

  \text{if (token.state ? \{ ACCEPT, REJECT,\text{'}\text{'}\})}
  \text{token.restart} = \text{TRUE}

  \text{if (token.restart == TRUE)} \\
  \{ 
    W \leftarrow 1 
    C \leftarrow 00 
    \text{HdMrk} \leftarrow \text{FALSE} 
    \text{if (R.restart == TRUE)} 
      \text{token.restart} \leftarrow \text{FALSE}
  \}

  \text{if (token.state ? \{ ACCEPT, REJECT\})}
  \{ 
    O \leftarrow \text{Trans}[R.\text{state}]
    \text{token.state} \leftarrow \text{Initial state}
  \}

  \text{else} \\
  \{ 
    O \leftarrow \text{'}\text{'}
    \text{state} \leftarrow \text{'}\text{'}
  \}
if (C==11)
    token.B ← 1
else
    token.B ← 0
C ← C + 1
Send token to P_{n-1}

P_i (n > i > 1):
On receipt of token from P_{i-1}
{
    R ← token
    if ( HdMrk == TRUE )
    {
        token.state, W, Mov ← Trans[R.state][W]
    }
    if ( R.HdEnable == TRUE )
    {
        HdMrk ← TRUE
        token.HdEnable ← FALSE
        Mov ← STAY
    }
    if ( token.restart == TRUE )
    {
        W ← I
        HdMrk ← FALSE
        C ← 00
    }
    send token to P_{i+1}
}
On receipt of token from P_{i+1}
{
    R ← token
    if ( HdMrk == TRUE )
    {
        token.state, W, Mov ← Trans[R.state][W]
    }
    if ( R.HdEnable == TRUE )
    {
        HdMrk ← TRUE
        token.HdEnable ← FALSE
        Mov ← STAY
    }
    if ( token.restart == TRUE )
    {
        W ← I
        HdMrk ← FALSE
        C ← 00
    }
    send token to P_{i-1}
}
The above algorithm simulates a Turing Machine under fault free execution, as is evident from the algorithm. The fact that it can recover from transient faults is due to the following facts:

- The movement of the token is hardcoded into the system and is independent of any corruptible data. The communication pattern is fixed.
- The system detects that at least once a state has been repeated in a computation when the counter overflows. It then resets itself and starts all over again.
- If the system accepts or rejects the string, it resets itself and restarts what it was doing, but in such a case none of the output registers \( O_i \) will change and the system would stabilize to the correct output under fault free execution.

5.4 Tradeoffs in Self-Stabilizing Algorithms

What really does a system sacrifice when it is made self-stabilizing? The two immediate performance degradations are in terms of memory and speed.

From the above algorithm, we see that if we are to only increase the memory required by a constant factor, the time required increases by a factor of the number of processes. Whereas a non-stabilizing simulation would have required \( O(\text{order}) \) time \([\text{order} = \text{time required by the algorithm}]\), the self stabilizing simulation requires \( O(n \times \text{order}) \) time \([n = \text{no. of processes}]\).

In general, under any self stabilizing algorithm, any local state (global state) consists of a primary portion (pertaining to input, processing and output variables required under fault free execution) and a secondary portion (pertaining to variables required specifically for self stabilization). For asymmetric code, the lower bound for the secondary portion is \( O(\log n) \) – for storing individual process ids. However, for probabilistic algorithms, even this is not a bound, though it has to be capable of generating random inputs.

From the literature, it appears that increased time is much more of an issue than increased space. Increase in memory rarely solves the problem of transient faults, as the entire memory may be corrupted by a fault.
6. Conclusion

6.1 Feedback systems

Self-stabilizing systems are part of a larger paradigm of systems – feedback systems. Feedback systems are defined as follows: Any system where a part of the output is fed back to the system as input is called a feedback system. Feedback systems are inherently recursive, and difficult to analyze. Self stabilizing systems are those feedback systems which aim to bound some of the parameters in their state space. There are three kinds of feedback systems:

1. Convergent Feedback Systems: The system converges to either a fixed point or a period if the feedback pushes the system in the direction of the attractor.
2. Divergent Feedback Systems: The system diverges to infinity in terms of its parameters if the feedback pushes it away from all the attractors.
3. Chaotic and Probabilistically Bounded Feedback Systems: The system never converges to any attractor but remains bounded.

Any finite state deterministic system is necessarily of the first kind, as the other two kinds of system demand an infinite state space.

Self-stabilizing systems are of the first or third kind. Self-stabilizing systems may be of the third kind only if the set of legitimate states is subsumed by the bounded attraction zone and the system is infinite state or probabilistic.

6.2 Beyond Distributed Systems

The concept of self-stabilizing systems is present outside the domain of distributed systems as well. It is a powerful formal model for dynamic convergent systems. Self-stabilization techniques have been applied on complex emergent systems which exhibit corrective and fixed behavior. Such widely varying systems include animal locomotion, human brain, cellular automata and the biosphere.

A case in point is the Daisyworld – a planet populated only by white and black daisies. [19] This simplistic model is to understand the stability of the biosphere. The white daisies have low radiation absorption capacity and thrive in high radiation zones, while black daisies have high radiation absorption capacity and thrive in low radiation zones. Depending on externally incumbent radiation, the temperature of the planet should change when the system is simulated. However, this is not the case, for the black and white daisies rearrange themselves along the poles and equator to always absorb a constant amount of heat and keep the temperature bounded within certain limits of incumbent radiation. For low incumbent radiation, black daisies congregate at the equator and the poles are not amenable to life. At medium incumbent radiation, black
daisies thrive at the poles, and white daisies at the equator. At high incumbent radiation, only white daisies thrive at the poles and the equator is not amenable to life. Such models are typically simulated and analytically explored using domain specific knowledge and self-stabilization.

Figure 6.1: The self-stabilizing parameters of temperature and area cover in Daisyworld

6.3 New Vistas and Emerging Fields

Fault-handling mechanisms which have so far been developed have a cost which grows with the size of the network. Fault mending provides a way to keep the cost dependent only on the number of failed nodes and not on the network size, and allows the remaining part of the network to carry on normal execution. [20] Another research area in self-stabilization is time-adaptive self-stabilization. The basic idea is to use a standard persistent bit algorithm, where the system is supposed to show tolerance to transient failures via replication, and convert that into a self-stabilizing persistent bit algorithm. k-faulty systems can thus be effectively stabilized for k < n/2. [21]
7. References


