

Lecture 17

Self-stabilization



Rajiv Misra

Dept. of Computer Science &
Engineering

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture, we have discussed about **'Message ordering, Group communication and Application level multicast algorithms'**

Content of this Lecture:

- In this lecture, we will discuss the **'concept of self-stabilization', 'related issues in the design of self-stabilizing algorithms and systems'** and **'Dijkstra's self-stabilizing token ring system'**.

Introduction: Concept of Self Stabilization

- The **idea of self-stabilization** in distributed computing was **first proposed by Dijkstra in 1974**.
- **The concept of self-stabilization** is that, regardless of its **initial state**, the system is **guaranteed to converge** to a **legitimate state** in a **bounded amount of time** by itself without any outside intervention. A **non-self-stabilizing system** may never reach a **legitimate state** or it may reach a legitimate state only temporarily.
- The main complication in designing a self-stabilizing distributed system is that nodes **do not have a global memory** that they can access instantaneously. Each node must make decisions based on the **local knowledge** available to it and actions of all nodes must achieve a **global objective**.

Contd...

- **The definition of legitimate and illegitimate states** depends on the particular application. Generally, all illegitimate states are defined to be those states which are not legitimate states.
- **Dijkstra** also gave an example of the concept of self-stabilization using a **self-stabilizing token ring system**. For any given **token ring** when there are **multiple tokens or there is no token**, then such global states are known as **illegitimate states**.
- When we consider a distributed system where a large number of systems are widely distributed and communicate with each other using message passing or shared memory approach, there is a possibility for these systems **to go into an illegitimate state**, for example, if a **message is lost**. The **concept of self-stabilization can help us recover from such situations** in distributed system.

Contd...

- Let us explain the **concept of self-stabilization** using an example. Let us take a group of children and ask them to stand in a circle.
- After few minutes, you will get an almost perfect circle without having to take any further action.
- In addition, you will discover that the shape of this circle is stable, at least until you ask the children to disperse.
- If you force one of the children out of position, the others will move accordingly, moving the entire circle in another position, but keeping its shape unchanged.

Contd...

- In this example, the group of children build a self-stabilizing circle: if some thing goes wrong with the circle, they are able to rebuild the circle by themselves, without any external intervention.
- The **time required for stabilization varies from experiment to experiment**, depending on the (random) initial position. However, if the field size is limited, this time will be bounded.
- The algorithm does not define the position of the circle in the field and so it will not always be the same. The position of each child relative to each other will also vary.

Contd...

- The **self-stabilization principle** applies to any system built on a significant number of components which are evolving independently from one another, but which are cooperating or competing to achieve common goals.
- This applies, in particular, to **large distributed systems** which tend to result from the **integration of many subsystems** and components developed separately at earlier times or by different people.

Contd...

- In this lecture, we will first present the **system model** of a distributed system and present **definitions of self-stabilization**.
- Next, we will discuss **Dijkstra's seminal work** and use it to motivate the topic and then
- We will discuss the **issues arising** from the Dijkstra's original presentation as well as several related issues in the design of self-stabilizing algorithms and systems.

System Model

- The **term distributed system** is used to describe set of computers that communicate over network. Variants of distributed systems have similar fundamental coordination requirements among the communicating entries, whether they are computers, processors or processes.
- Thus an **abstract model** that ignores the specific settings and captures the important characteristics of a distributed system is usually used.
- In a distributed system, each computer runs a program composed of executable statements. Each execution changes the content of the computer's logical memory.
- An **abstract way to model a computer** that executes a program is to use the **state machine model**.

Contd...

- A distributed system model comprises of a set of **n state machines** called processors that communicate with each other.
- We usually denote the ***i*th** processor in the system by **P_i** . Neighbors of a processor are processors that are directly connected to it.
- A processor can directly communicate with its neighbors. A distributed system can be conveniently represented by a graph in which each processor is represented by a node and every pair of neighboring nodes are connected by a link.

Contd...

- The **communication between neighboring processors** can be carried out either by **message passing or shared memory**.
- Communication by writing in and reading from the shared memory usually fits systems with processors that are geographically close together, such as multiprocessor computer.
- A message-passing distributed model fits both processors that are located close to each other as well as that are widely distributed over a network.
- In the message-passing model, neighbors communicate by sending and receiving messages.
- Message passing communication model: **queue(s) Q_{ij} , for messages from P_i to P_j**

Contd...

- It is convenient to identify the state of a computer or a distributed system at a given time, so that no additional information about the **past of the computation** is needed in order to **predict the future behavior** (state transitions) of the computer or the distributed system.
- A full description of a message passing distributed system at a particular time consists of the state of every processor and the content of every queue (messages travelling in the communication links).

Contd...

- The term system configuration (or configuration) is used for such a description.

A configuration is denoted by

$c = (s_1, s_2, \dots, s_n, q_{1,2}, q_{1,3}, \dots, q_{i,j}, \dots, q_{n,n-1}),$

where $s_i, 1 \leq i \leq n$ is the state of P_i and $q_{i,j}$,

$i \neq j$ is the state of queue $Q_{i,j}$, that is, messages sent by P_i to P_j but not yet received.

- The behavior of a system consists of a set of states, a transition relation between those states, and a set of fairness criteria on the transition relation.

Contd...

- The system is usually modeled as a **graph of processing elements** (modeled as state machines), where edges between these elements model unidirectional or bidirectional communication links.
- Let **N be an upper bound on n (the number of nodes in the system)**. The communication network is usually restricted to the neighbors of a particular node.
- Let **$\bar{g}$ denote the diameter of the network** (i.e., the length of the longest unique path between two nodes) and **let Δ denote the upper bound on $\bar{g}$.**

Contd...

- A **network is static** if the communication topology remains fixed. It is dynamic if links and network nodes can go down and recover later.
- In the **context of dynamic systems**, self-stabilization refers to the time after the “final” link or node failure. The term “final failure” is typical in the literature on self-stabilization.
- Since stabilization is only guaranteed eventually, the assumption that faults eventually stop to occur implies that there are no faults in the system for “sufficiently long period” for the system to stabilize.
- In any case, it is assumed that the topology remains connected, i.e., there exists a path between any two nodes.

Contd...

- In the **shared memory model**, processors communicate using shared communication registers (hereafter, called registers). Processors may write in a set of registers and may read from a possibly different set of registers.
- Two neighboring nodes have **access to a common data structure, variable or register** which can store a certain amount of information. These variables can be distinguished between input and output variables (depending on which process can modify them).
- When executing a step, a process may read all its input variables, perform a state transition and write all its output variables in a single atomic operation. This is called **composite atomicity**. A weaker notion of a step (**called read/write atomicity**) also exists where a process can only either read or write its communication variables in one atomic step.

Contd...

- The configuration of a system with **n processors and m communication registers is denoted by**

$$c = (s_1, s_2, s_3 \dots s_n, r_1, r_2 \dots r_m),$$

where s_i , $1 \leq i \leq n$, is the state of P_i and

r_j , $1 \leq j \leq m$, is the contents of a communication register.

Contd...

- Algorithms are **modeled as state machines performing a sequence of steps**. A step consists of reading input and the local state, then performing a state transition and writing output.
- Communication can be by exchanging messages over the communication channels. An algorithm may be randomized, i.e., have access to a source of randomness (a random number generator or a random coin flip). If an algorithm is not randomized, we will call it deterministic.
- A related characteristic of a system model is its execution semantics. In self-stabilization, this has been encapsulated within the notion of a **scheduler or daemon** (also demon).
- Under **a central daemon, at most one processing element is allowed** to take a step at the same time.

Definition of Self-stabilization

- We have seen an informal definition of self-stabilization at the beginning.
- **Formally, we define self-stabilization** for a **system S** with respect to a **predicate P** over its set of global states, where P is intended to identify its correct execution. States satisfying P are called **legitimate states** and those not satisfying P are called illegitimate states. We use the terms safe and unsafe interchangeably with legitimate and illegitimate, respectively. A system S is self-stabilizing with respect to predicate P if it satisfies the following two properties:
 - (i) **Closure:** P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.
 - (ii) **Convergence:** Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

Contd...

- **Arora and Gouda** introduced a more generalized definition of self-stabilization, called **stabilization**, which is defined as follows.
- We define stabilization for a system S with respect to two predicates P and Q , over its set of global states. Predicate Q denotes a restricted start condition. **S satisfies $Q \rightarrow P$ (read as Q stabilizes to P)** if it satisfies the following two properties:
 - (i) **Closure** P is closed under the execution of S . That is, once P is established in S , it cannot be falsified.
 - (ii) **Convergence** If S starts from any global state that satisfies Q , then S is guaranteed to reach a global state satisfying P within a finite number of state transitions.
- Note that **self-stabilization is a special case of stabilization** where Q is always true, that is, if S is self-stabilizing with respect to P , then this may be restated as **$TRUE \rightarrow P$ in S .**

Contd...

Reachable Set:

- Often when a programmer writes a program, he/she does not have a particular definition of safe and unsafe states in mind but develops the program to function from a particular set of start states.
- In such situations, it is reasonable to define as **safe those states that are reachable under normal program execution from the set of legitimate start states**. These states are referred to as the reachable set.
- So, when we say that a program is self-stabilizing without mentioning a predicate, we mean with respect to **the reachable set**. By definition, **the reachable set is closed under program execution**, and it corresponds to a **predicate over the set of states**.

Contd...

Transient failure:

- A **transient failure is temporary (short lived)** and it does not persist.
- A transient failure may be caused by **corruption of local state of processes** or by **corruption of channels or shared memory**.
- A transient failure **may change the state of the system**, but not its behavior.

Issues in the design of self-stabilization algorithms

- A distributed system **comprises of many individual units** and many issue arise in the design of self-stabilization algorithms in distributed system. Some of the **main issues** are as follows:
- Number of states in each of the individual units in a distributed system.
- Uniform and non-uniform algorithms in distributed systems.
- Central and distributed demon.
- Reducing the number of states in a token ring.
- Shared memory models.
- Mutual exclusion.
- Costs of self-stabilization.

Dijkstra's self-stabilizing token ring system

- The mentioned issues can be explained with the help of **Dijkstra's landmark self-stabilizing token ring system.**
- His system consisted of a **set of n finite state machines connected in the form a ring.** He defines a privilege of a machine to be the ability to change its current state.
- This ability is based on a Boolean predicate that consists of its current state and the states of its neighbors. When a machine has a privilege, it is able to change its current state, which is referred to as a move.
- Furthermore, when multiple machines enjoy a privilege at the same time, the choice of the machine that is entitled to make a move is made by a central demon, which arbitrarily decides which privileged machine will make the next move.

Contd...

A legitimate state must satisfy the following constraints:

1. There must be at least one privilege in the system (**liveness or no deadlock**).
2. Every move from a legal state must again put the system into a legal state (**closure**).
3. During an infinite execution, each machine should enjoy a privilege an infinite number of times (**no starvation**).
4. Given any two legal states, there is a series of moves that change one legal state to the other (**reachability**).

Contd...

- **Dijkstra** considered a **legitimate (or legal) state** as one in which exactly one machine enjoys the privilege.
- This corresponds to a **form of mutual exclusion**, because the **privileged process** is the only process that is allowed in its critical section.
- Once the process leaves the critical section, **it passes the privilege to one of its neighbors.**

The number of states in each of the individual units

- The number of states that each machine must have for the self-stabilization is an important issue.
- **Dijkstra offered three solutions for a directed ring with n machines**, $0, 1, \dots, n-1$, each having **K states**, (i) $K \geq n$, (ii) $K = 4$, (iii) $K = 3$.
- It was later proven by **Ghosh** that a **minimum of three states is required in a self-stabilizing ring**.
- In all three algorithms, **Dijkstra** assumed the existence of **at least one exceptional machine** that behaved differently from the others.

First solution ($K \geq n$)

- For any machine, we use the symbols **S**, **L**, and **R** to denote its own state, the state of the left neighbor and the state of the right neighbor on the ring, respectively.

The exceptional machine:

If $L = S$ then

$S := (S+1) \bmod K$

End If;

The other machines:

If $L \neq S$ then

$S := L$

End If;

Contd...

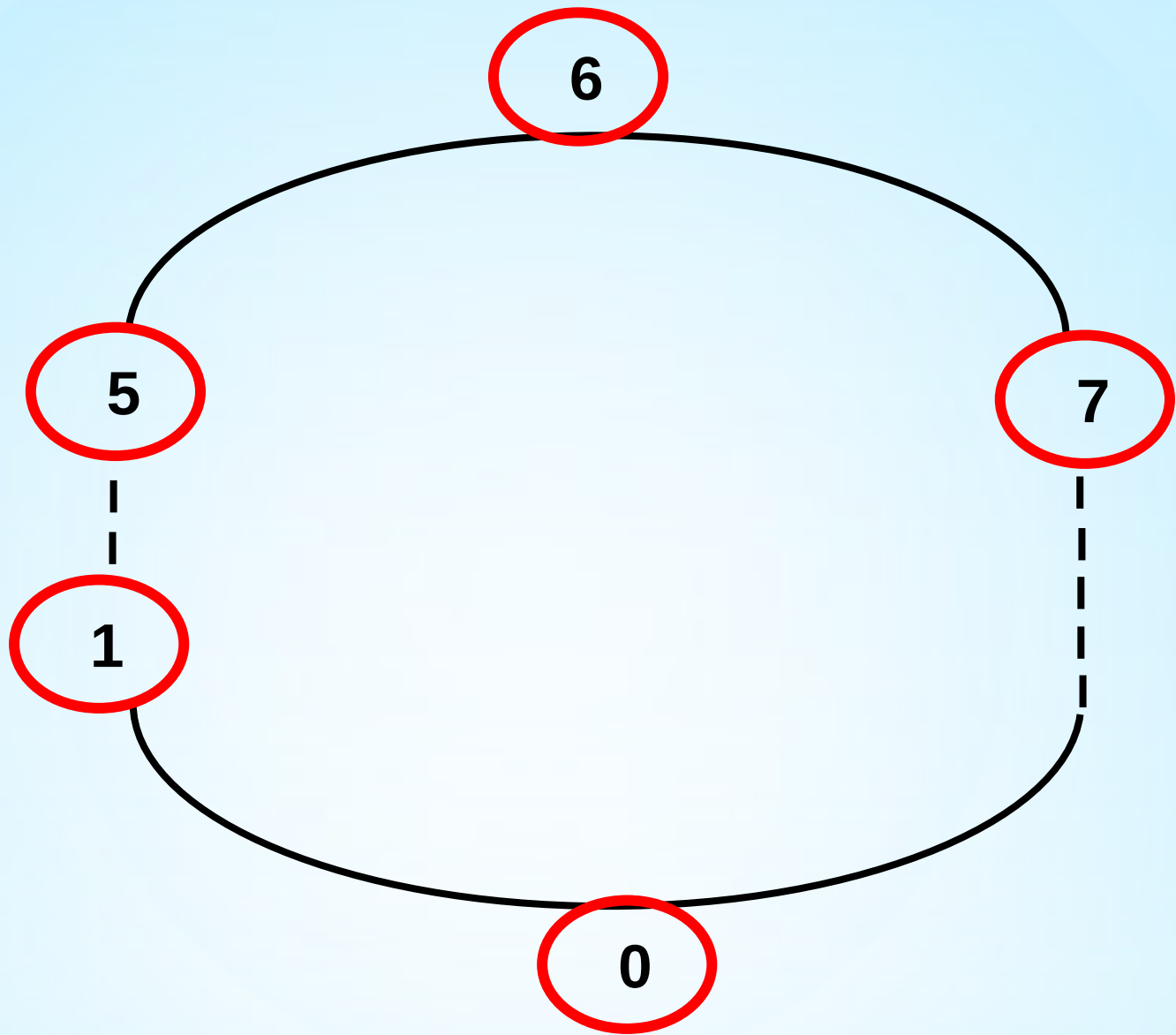
- In this algorithm, except the **exceptional machine(machine 0)**, all other machines follow the same algorithm. **In the ring topology, each machine compares its state with the state of the anti-clockwise neighbor** and if they are not same, it updates its state to be the same as that of its anti-clockwise neighbor.
- So, if there are **n machines** and each of them is initially at a **random state $r \in K$** , then all the machines (except the exceptional machine, **machine 0**) whose states are not the same as their anti-clockwise neighbor are said to be privileged and there is a central demon that decides which of these privileged machines will make the move.

Contd...

- Suppose **machine 6** (assume $n \gg 6$) makes the first move. It is obvious that its state is not the same as that of **machine 5** and hence it had the privilege to make the move and finally sets its state to be the same as that of **machine 5**. Now **machine 6** loses its privilege as its state is same as that of its anti-clockwise neighbor (**machine 5**).
- Next, suppose **machine 7**, whose state is different from the state of **machine 6**, is given the privilege. It results in making the state of **machine 7** the same as that of **machine 6**. Now **machines 5, 6, and 7** are in the same state.
- Eventually, all the machines will be in the same state in the similar manner. At this point, only the exceptional machine (**machine 0**) will be privileged as its condition $L = S$ is satisfied, i.e., its state is the same as that of its anti-clockwise neighbor.

Contd...

- Now there exists only one privilege or token in the system (at **machine 0**). **Machine 0** makes a move and changes its state **from S to $(S+1) \bmod K$** . This will make the next machine, **machine 1**, privileged as its state is not the same as its anti-clockwise neighbor, i.e., **machine 0**. Thus, it can be interpreted as the token is currently with **machine 1**.
- **Machine 1**, as per the algorithm, changes its state to the same state as that of **machine 0**. This will move the token to **machine 2** as its state is now not same as that of **machine 1**. Likewise, the token keeps circulating around the ring and the system is stable.
- This is a simple algorithm, but it requires a **number of states**, which **depends on the size of the ring**, which may be awkward for some applications.



Second solution ($K = 3$)

- The second solution uses only **three-state machines** and is presented in **Algorithm 17.1**
- The state of each machine is in $\{0, 1, 2\}$.
- In the first algorithm, there is only one exceptional machine, machine 0.
- In the second solution, there are two such machines, **machine 0**, referred to as the bottom machine, and **machine $n-1$** , referred to as the top machine.

Contd...

The bottom machine, machine 0:

If $(S+1) \bmod 3 = R$ then
 $S := (S-1) \bmod 3$

The top machine, machine $n-1$:

If $L = R$ and $(L+1) \bmod 3 \neq S$ then
 $S := (L+1) \bmod 3$

The other machines:

If $(S+1) \bmod 3 = L$ then
 $S := L$
If $(S+1) \bmod 3 = R$ then
 $S := R$

Contd...

- In this algorithm, the bottom machine, machine 0, behaves as follows:

If $(S+1) \bmod 3 = R$ then

$S := S-1 \bmod 3$

- Thus, the state of the bottom machine depends upon its current state and the state of its right neighbor.
- The condition $(s+1) \bmod 3$ covers the three possible states; for $s = 0, 1, 2$, we have $(s+1) \bmod 3 = 1, 2, 0$. These result in the following three possibilities:
 1. If $s = 0$ and $r = 1$, then the state of s is changed to 2.
 2. If $s = 1$ and $r = 2$, then the state of s is changed to 0.
 3. If $s = 2$ and $r = 0$, then the state of s is changed to 1.

Contd...

- The top machine, **machine n-1**, behaves as follows:

If $L = R$ and $(L+1) \bmod 3 \neq S$ then

$S := (L+1) \bmod 3$

- The state of the top machine depends upon both its left and right neighbors (the bottom machine). The condition specifies that the **left neighbor (L)** and the **right neighbor (R)** should be in the same state and **$(L+1) \bmod 3$** should not be equal to S . (Note that **$(L+1) \bmod 3$ is 1, 2, 0 when L is 0, 1, 2, respectively**) Thus, the state of the top machine is as follows:
 1. **1**, when its left neighbor is 0.
 2. **2**, when its left neighbor is 1.
 3. **0** when its left neighbor is 2.

Contd...

- All other machines behave as follows:

If $(S+1) \bmod 3 = L$ then

$S := L$

If $(S+1) \bmod 3 = R$ then

$S := R$

- While finding out the state of the other machines (**machines 1 and 2** in the example below), we first compare the state of a machine with its left neighbor:

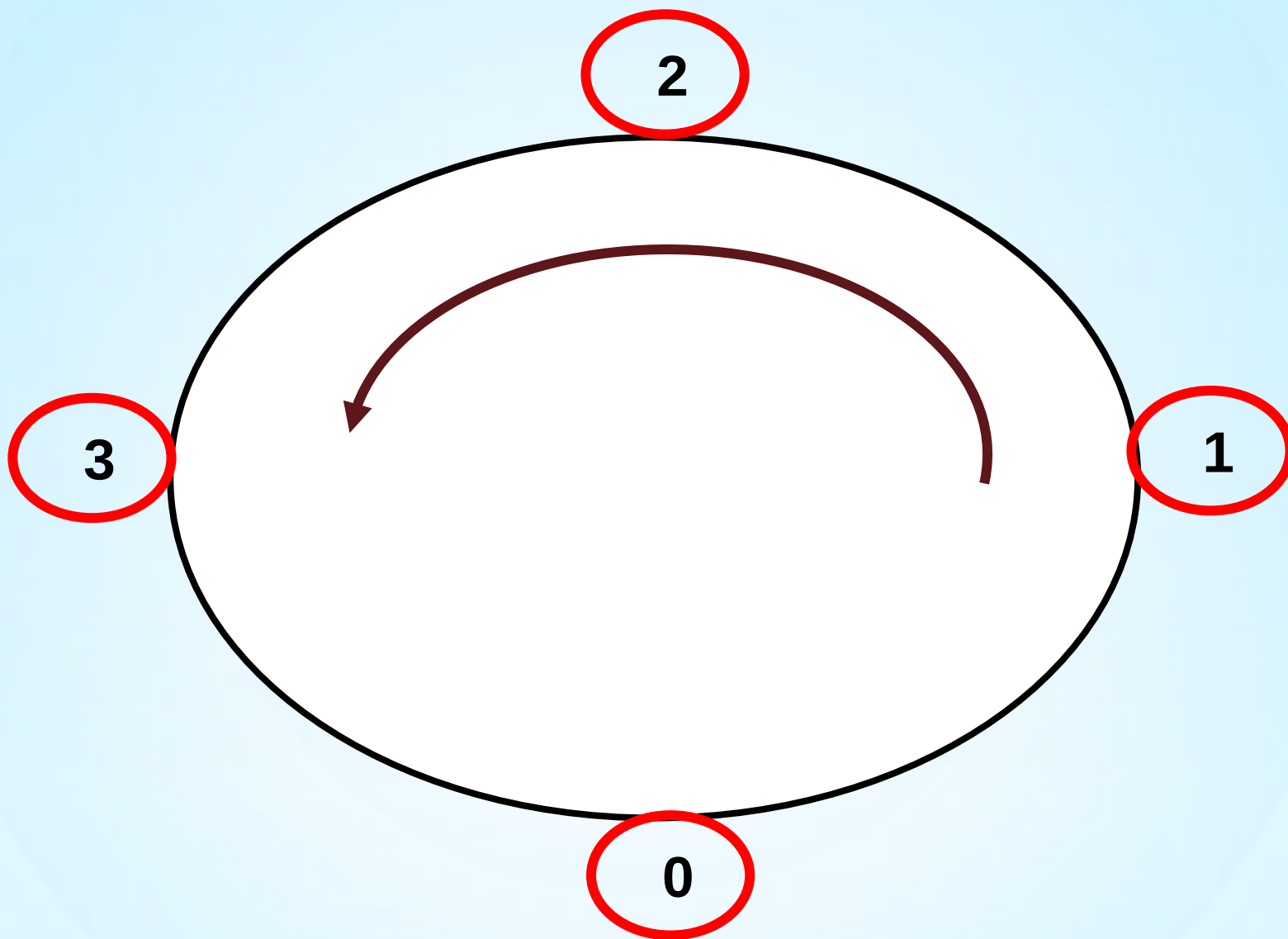
1. **If $s = 0$ and $L = 1$** , then $s = 0$.
2. **If $s = 1$ and $L = 2$** , then $s = 2$.
3. **If $s = 2$ and $L = 0$** , then $s = 1$.

Contd...

- If the given conditions are not satisfied, then the machine compares its state with its right neighbor.
- A sample execution of **Dijkstra's three-state algorithm for a ring of four processes (0, 1, 2, 3)** is shown in **Table 17.1. Machine 0 is the bottom machine and machine 3 is the top machine.**
- The last column in the table gives the number of the machine chosen to make the next move.
- Initially, three privileges exist in the system. The number of privileges decreases until only one privilege is left in the system.

Table 17.1 An example execution of Dijkstra's three-state algorithm

State of machine 0	State of machine 1	State of machine 2	State of machine 3	Privileged machines	Machine to make move
0	1	0	2	0, 2, 3	0
2	1	0	2	1, 2	1
2	2	0	2	1	1
2	0	0	2	0	0
1	0	0	2	1	1
1	1	0	2	2	2
1	1	1	2	2	2
1	1	2	2	1	1
1	2	2	2	0	0
0	2	2	2	1	1
0	0	2	2	2	2
0	0	0	2	3	3
0	0	0	1	2	2



Observations

- We can make the following observations:
 1. There are **no deadlocks** in any state (at least one privilege is present).
 2. The **closure property** is satisfied (the system moves from a legal state to a legal state).
 3. **No starvation** (each machine has a chance of making more than 1 move).
 4. **Reachability** (there are always a series of moves to reach from one legal state to other).
- **All four constraints for a legitimate state are satisfied. So the system is stabilized.**

Few Other Works in the Field of Self Stabilization

Author	Year	Algorithm
E.W. Dijkstra	1974	Proposed the idea of self-stabilization
E. J. H. Chang et al.	1987	Costs of self-stabilization.
S. Aggarwal	1993, 1994	Time optimal self-stabilizing spanning tree algorithms
A. Arora and M. G. Gouda	1994	Distributed reset
G. Antonioiv and P.K. Srimani	1995, 1997, 1998	Self-stabilizing algorithms for minimum spanning trees construction
H. Kakugawa	2000	Universal Self-Stabilizing Mutual Exclusion Algorithm
F.C. Gartner	2003	Survey of algorithms for construction of self-stabilizing spanning trees.
Z. Shi et al.	2004	An anonymous self-stabilizing algorithm for 1-maximal independent set in trees.

Conclusion

- **Self-stabilization** has been used in many areas and the areas of study continue to grow. Algorithms have been developed **using central or distributed demons** and **uniform and non-uniform networks**.
- The algorithms that assume a central demon can usually be easily extended to support distributed demon, so these algorithms are still useful when applied to distributed systems.
- In this lecture, we have discussed the '**concept of self-stabilization**', '**system model**', '**related issues in the design of self-stabilizing algorithms and systems**' and also discuss '**Dijkstra's self-stabilizing token ring system**'