1-1-1993

Self-stabilizing deadlock algorithms in distributed systems

Mitchell Elliott Flatebo

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/289

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Self-stabilizing deadlock algorithms in distributed systems

Flatebo, Mitchell Elliott, M.S.
University of Nevada, Las Vegas, 1993
SELF-STABILIZING DEADLOCK
ALGORITHMS IN DISTRIBUTED SYSTEMS

by

Mitchell Elliott Flatebo

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May 1993
The thesis of Mitchell Elliott Flatebo for the degree of Master of Science in Computer Science is approved.

Chairperson, Ajoy Kumar Datta, Ph.D

Examiner Committee Member, Kazem Taghva, Ph.D

Examiner Committee Member, Laxmi Gewali, Ph.D

Graduate Faculty Representative, Ashok Iyer, Ph.D

Graduate Dean, Ronald W. Smith, Ph.D

University of Nevada, Las Vegas
May 1993
ABSTRACT

A self-stabilizing system is a network of processors, which, when started from an arbitrary (and possibly illegal) initial state, always returns to a legal state in a finite number of steps. Self-stabilization is an evolving paradigm in fault-tolerant computing. This research will be the first time self-stabilization is used in the areas of deadlock detection and prevention. Traditional deadlock detection algorithms have a process initiate a probe. If that probe travels around the system and is received by the initiator, there is a cycle in the system, and deadlock is detected. In order to prevent deadlocks, algorithms usually rank nodes in order to determine if an added edge will create a deadlock in the system. In a self-stabilizing system, perturbances are automatically dealt with. For the deadlock model, the perturbances in the system are requests and releases of resources. So, the self-stabilizing deadlock detection algorithm will automatically detect a deadlock when a request causes a cycle in the wait-for graph. The self-stabilizing prevention algorithm prevents deadlocks in a similar manner. The self-stabilizing algorithms do not have to be initiated by any process because the requests and releases create a perturbance which is dealt with automatically.
# Contents

Abstract ......................................................... iii
Acknowledgments ................................................. vi

1 INTRODUCTION ........................................... 1
1.1 Dijkstra's Model ........................................... 2
1.2 Deadlock Model ............................................ 3
1.3 Notation and Demons ...................................... 6

2 DEADLOCK DETECTION ................................. 8
2.1 Variables .................................................... 9
2.2 Single Outstanding Request Algorithm ............... 11
2.3 Multiple Outstanding Requests ......................... 14
2.4 Deadlock Resolution ..................................... 16
2.5 Conclusions ................................................ 17

3 DEADLOCK PREVENTION ............................. 19
3.1 Variables .................................................... 20
3.2 Deadlock Prevention Algorithm ....................... 20
3.3 Example of Algorithm .................................... 23
3.4 Modifications .............................................. 24
3.5 Conclusions ................................................ 25

4 CONCLUSIONS ........................................ 27
BIBLIOGRAPHY ............................................. 28
List of Figures

2.1 Three Cases for Single Outstanding Request Model. . . . . . . . . . 11

3.1 Example of Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Successor Sets for Processes Figure 3.1a . . . . . . . . . . . . . . . . 23
3.3 Successor Sets for Processes in Figure 3.1b . . . . . . . . . . . . . . . 24
ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Ajoy Kumar Datta for helping and encouraging me throughout the course of this thesis. I was not always the easiest student to handle, but he was there as my advisor and more importantly as a friend. I would also like to thank the Computer Science department for allowing me to be a teaching assistant. Special thanks to Dr. Kazem Taghva, Dr. Laxmi Gewali, and Dr. Ashok Iyer for serving on my committee.

This thesis is dedicated to my family. My sister, Marie, and mother, Jerry, who supported and encouraged me while I was completing my undergraduate degrees and my first graduate degree. My brothers, Martin and Mike, who made me realize the importance of education. And last, but certainly not least, I would like to thank my wife, Maureen, for being there for me while I was completing my research.
Chapter 1

INTRODUCTION

A distributed system consists of a set of loosely connected machines which do not share a global memory. Depending on the way the machines are connected in the network and the time it takes for two machines to communicate with each other, each machine gets a partial view of the global state.

A fundamental criterion in the design of robust distributed systems is to embed the capability of recovery from unforeseen perturbances. While most of the existing systems cater to permanent failures by introducing redundant components, the issue of transient failures is often ignored or inadequately addressed. Considering the computation in a distributed system to be a totally or partially ordered sequence of states in the state space, it is conceivable to encounter a transient malfunction due to message corruption, sensor malfunction or incorrect read/write memory operations, that transforms the global state of the system into an illegal state, from which recovery is not guaranteed. Examples are token-ring networks in which the token is lost or duplicate tokens are generated, or sliding window protocols in which the window alignment is lost due to transient errors. The essence of these examples is that if the set of possible global states of a distributed system is partitioned into legal and illegal states, then transient failures can potentially put the system into an illegal state, which may continue indefinitely unless it is externally detected and suitable
corrective measures are taken. A self-stabilizing system guarantees that regardless of the current state, the system is guaranteed to recover to a legal configuration in a finite number of steps and remain in the legal configuration thereafter, until a subsequent malfunction occurs. This property makes the system more robust. No startup or initialization procedure needs to be used because the system stabilizes by itself. If one machine fails and restarts, its local state may cause an illegal global state, but the system will correct itself in a finite amount of time. The ability of the system to correct certain errors without outside intervention makes a self-stabilizing system more reliable and more desirable than systems that are not self-stabilizing.

The notion of self-stabilization has been prevalent in the field of mathematics and control theory for many years. Consider for example the Newton-Raphson method of finding the square root of a number where, regardless of what estimate is made about the initial value of the square root, the solution converges to the desired value in a finite number of steps. Similar notions have been used in feedback control systems for many decades. In the field of distributed systems, the study of self-stabilization was pioneered by Dijkstra [4], and has received considerable attention in recent years.

This research presents the state-of-the-art in the design of self-stabilizing distributed systems. The focus of this research will be in the area of deadlock detection and prevention. Traditional deadlock detection and prevention algorithms have nodes initiate a probe of some sort. This initiation is not necessary when self-stabilization is used. If a deadlock occurs, it is automatically detected (or prevented) by the algorithm. No outside intervention is necessary.

1.1 Dijkstra's Model

In 1974, Dijkstra introduced the property of self-stabilization in distributed systems [4]. His system consisted of a set of \( n \) finite state machines connected in the form of 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 depends on 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 \textit{move}.

A system is called self-stabilizing when, regardless of the initial state and regardless of the privilege selected each time, the system always converges to a legal configuration in a finite number of steps. 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.

The legal states must satisfy the following properties:

\begin{itemize}
\item [P1] There must be at least one privilege in the system (no deadlock).
\item [P2] Every move from a legal state must again put the system into a legal state (closure).
\item [P3] During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation).
\item [P4] Given any two legal states, there is a series of moves that change one legal state to the other (reachability).
\end{itemize}

Dijkstra considered a legal state as one in which exactly one machine enjoys a 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. This characterization of legal states has been used in many of the early papers [1] on self-stabilization, but this research uses the idea of self-stabilization in a new area, deadlock detection and prevention.

\subsection{1.2 Deadlock Model}

The traditional deadlock prevention model is a distributed database (DDB). A DDB consists of resources, controllers, and processes. Each controller manages a set of resources and a set of processes. A process requests resources through its controller. If the resource requested is not controlled by the local controller, the controller communicates with the controller that manages the resource. These requests made by
processes in the system may have to wait because the resource is not available. If process \( i \) requests a resource which is held by process \( j \), this waiting is denoted by an edge in a graph from process \( i \) to process \( j \). If a process releases a resource, the graph is again changed. This graph is called a wait-for graph, and it is a way of describing all the dependencies in the system. A set of processes is deadlocked when no process in the set can execute because each process requires a resource held by some other process in the set. This deadlock corresponds to a cycle in the wait-for graph.

There are a number of reasons why distributed deadlock detection seems more attractive than a centralized scheme. A centralized scheme is one in which a single agent (process) is responsible for deadlock detection, while in the distributed scheme, no single site knows the resource requirements of the entire system. The centralized scheme is vulnerable to failures of the central detector. Once this central detector fails, it results in long delays as a new central detector is determined and supplied with the up-to-date wait-for information. Also, due to the heavy traffic to and from the central detector, it constitutes a performance bottleneck, limiting the performance of the database system.

The above model is used here. However, the controllers are not mentioned in the algorithms or proofs of correctness for simplicity. The processes and the states of the processes are used. The states of the processes depend on local variables. Each process is thought to have the information of its own state and its neighbor states. These states are given as variables. These variables are changed according to the requesting and releasing of resources. By examining the states of it neighbors, each process in the system will eventually determine if it is in a possible deadlock (once the system is stabilized).

There are two models of resource requests. The simple model is when there are only single outstanding requests. When a process makes a request for a resource in this model, it must wait until it receives this resource or no longer wants the resource until it can make another request. When using the single outstanding request model, each process in the wait-for graph can have at most one outgoing edge (it may have several incoming edges). The first algorithm discussed in Chapter 2, uses this model.
Because of the simplicity of this model, it is not very useful. The more general model allows multiple outstanding requests. So, each process in the wait-for graph can have any number of incoming and outgoing edges. The second detection algorithm in Chapter 2 and the prevention algorithm in Chapter 3 use this model.

There are three ways of handling deadlocks - detection, prevention and avoidance. Deadlock detection is the approach in which a deadlock is allowed to occur. Routines check for the presence of deadlock and steps are taken to break the deadlock if one exists, generally by aborting a process, canceling all its request messages and releasing all resources it currently holds. A number of algorithms have been proposed for detecting deadlocks in distributed systems [3, 8, 9]. In distributed database system, the problem is to find cycles in a distributed wait-for graph, where no single process knows the entire graph. Some algorithms detect deadlocks by first constructing and then finding cycles in the transaction wait-for graph (a directed graph where nodes represent transactions and edges represent the wait-for relationships) while some others use a probe technique. Probes are special messages used to detect the cycles. Probes follow the edges of the wait-for graph to search for a cycle. The self-stabilizing algorithms discussed here do not use either of these techniques. Each process examines the states of its neighbors in the wait-for graph and eventually determines if it is deadlocked or not.

In deadlock prevention, the system is designed in such a way that deadlock can never occur, which is taken care of by making sure that the necessary and sufficient conditions for deadlock are never met. The basic idea of deadlock prevention is to restart a process if the system finds that it will cause a deadlock. The method of pre-allocating all the requested resources is no longer feasible as the processes are data dependent. Hence, it is quite difficult to request the resources, as the required resources are not known a priori. Even for the designer of these deadlock prevention algorithms, it is very hard to be sure that the system will really be deadlock-free, as possible deadlocks can be easily overlooked when reasoning informally about a system. Time-stamp based synchronization techniques can be used as a method of deadlock prevention. The technique adopted for preventing deadlocks is based on
the notion of coloring the nodes of the wait-for graph and is built on a signalling mechanism which can be implemented on an underlying routing protocol. Again, the self-stabilizing algorithm for prevention discussed in Chapter 3 does not use this method. Other schemes such as the continuous ranking of nodes [7] can be used. However, there is still some initiation of the algorithm which is required, but the self-stabilizing algorithm does not require this initiation.

In deadlock avoidance, some knowledge of the future process behavior is used to constrain the resource allocation to avoid deadlock in the system. The algorithms discussed here have no such knowledge, so this deadlock handling technique is not discussed here.

There are several advantages to a self-stabilizing deadlock algorithms:

• The algorithm runs continually (no initiation of the algorithm needs to be done).
• Any resource request or release automatically creates privileges in the system. Once the system stabilizes, there will be no privileges (any deadlock will be detected or prevented).
• No initialization of the local variables needs to be done, because a self-stabilizing algorithm does not require any initialization.
• The statements in the algorithm can be executed in any order, and the system will still stabilize.
• The algorithm automatically tolerates transient errors (message loss, message corruption, etc.).

1.3 Notation and Demons

The program for each process has the form:

< statement >
< statement >
   ...
< statement >

Each statement has the form: < guard > → < action >
A guard is a boolean expression over the variables that a process can read (its own along with those in adjacent processes). For the algorithms, the adjacent processes of a process $i$ are its immediate predecessors and successors in the wait-for graph. If some process has a statement whose guard is true, then that process has a privilege and may make a move (execute the action). If several privileges exist in the system, the execution depends on the demon which is being used. Dijkstra assumed the presence of a central demon. This is the simplest of the four demons which are defined as follows:

1. **central demon:** Actions are executed atomically, one at a time. The central demon chooses one process from the set of privileged processes to make a move. No assumptions are made about this choice.

2. **randomized central demon:** Actions are executed atomically, one at a time. The central demon randomly chooses one process from the set of privileged processes to make a move.

3. **distributed demon:** Processes are allowed to move simultaneously. If a distributed demon is present, at any point in time, any subset of the set of privileged processes can move at this time.

4. **read/write demon:** Processors communicate through shared registers, and all of the shared registers are serializable with respect to read/write operations.

As the interleaving of the reads (in the guards) and the writes (in the assignment statements) changes, the behavior of the algorithm can change also. The central demon is more restrictive, hence it is easier to verify the algorithm. However, Burns et al. [2] developed a theory relating the correctness of an algorithm in the presence of a distributed demon to the correctness of that algorithm in the presence of a central demon. The randomized central demon is a special case of the central demon, so it is even more restrictive. The distributed demon is a special case of the read/write demon. The algorithms presented here work in the presence of any of the four demons.
Chapter 2

DEADLOCK DETECTION

The standard way of detecting deadlocks [3] is to send probes around the system. One process in the system initiates a probe, and if the initiator receives its own probe back, it declares a deadlock in the system. This chapter introduces a different scheme for detecting deadlocks [5]. Instead of using probes, states are used. For a given process \( i \), the state of \( i \) can be read by \( i \) and all of its neighbors (processes that depend on \( i \) or processes that \( i \) depends on). According to the states of its neighbors, a process can change its own state. These local states define a global state of the system which depends on these local states. The global states of the system are split up into legal and illegal states. The legal states of the algorithms discussed in this chapter are characterized by two criteria: (1) each process knows whether or not it is in a deadlock situation and (2) there are no privileges in the system. The algorithms in this chapter assume that each process decides on its own whether or not to make a move. This is equivalent to having a distributed demon present. The steps of the algorithm are not sequential. If a process is privileged by more than one step, it does not matter which move is executed.

This chapter presents two deadlock detection algorithms which use states to detect deadlocks rather than probes. Each process in the system has a certain state, and a process can determine if it is deadlocked by examining this state. The legal global
states in the system are defined by the knowledge of deadlocks. If the system is in a legal state, then all the processes that are deadlocked know that they are deadlocked, and all the processes that are not deadlocked do not declare that they are deadlocked.

Two algorithms are given in this chapter. The first algorithm detects deadlocks in a single outstanding request model. The second algorithm detects deadlocks in the multiple outstanding request model. The correctness proofs are given with the algorithms. Methods of resolution are also discussed for each algorithm.

2.1 Variables

The first algorithm. Detection Algorithm 1, presented in this chapter assumes that there are only single outstanding requests which means a process can only wait for one resource at a time. Detection Algorithm 1 uses three variables, source, dep, and deadlock. Each process in the wait-for graph can only have one outgoing edge, but it can have 0 or more incoming edges. The dep variable for process i contains the process identification number of a process which depends on i. In the algorithm, a process only changes its dep variable to a smaller value. When there are no privileges in the system, process i's dep variable will contain the lowest numbered process that depends (directly or indirectly) on i. A process i is a direct dependent of process j if there is an edge from i to j in the wait-for graph. If process i depends on process j but there is no edge between these processes in the wait-for graph, process i is an indirect dependent of process j. The source variable of process i contains the process id of the process that caused i to change its dep variable. This is used in order to determine a cycle in the wait-for graph. If a process receives the same information from two sources (it receives the same information from a neighbor different from source), then it declares a deadlock. The process id will be denoted by $P_i$ for process i. The deadlock variable is the variable used to determine whether or not a process is deadlocked. A process can have several direct dependents (their is an edge from the dependent to the process in the wait-for graph), and the dep variable for predecessor $k$ is denoted by $dep_k$. A process i's set of direct dependents is denoted as $PRED$ in
the algorithm. This can be seen in Figure 2.2. In this figure, the numbers inside the circle contain the value of the dep variable, and the small numbers outside the circle are the actual process ids. The shaded node is the node with the lowest process id. In Figure 2.2a, process 1's successor is 2, and its predecessors are 0 and 3. In the algorithm, $P_i$ is process $i$'s id. The dep variables in Figure 2.2 contain the values of the variables after the system is stable (no privileges exist).

The second algorithm, Detection Algorithm 2, uses a successor set to determine deadlock. This algorithm does not make any assumptions about the number of requests a process can make. Each process maintains a set called the Succ set. The successor set of process $i$, $Succ_i$, is defined as follows: $\forall j \in Succ_i, Succ_j \subseteq Succ_i$. Each process has a deadlock variable which is the same as in Detection Algorithm 1. If a process $i$ has $k$ immediate successors, then $Succ_1$, $Succ_2$, ..., $Succ_k$ are the successor sets for all successors of $i$ and $P_1$, $P_2$, ..., $P_k$ are the actual process ids of $i$'s successors. Both algorithms use deadlock. This denotes the deadlock variable of the process' successor. This is used to notify the processes that are not in the cycle that they depend on a deadlocked process.

In both algorithms, each request that is not granted causes a state change. This state change in the system causes privileges to be generated. Once there are privileges in the system, the system is in an illegal configuration and must converge to a state where no privileges exist. After this occurs, each process in the wait-for graph will know whether or not they are deadlocked. A process will know its status because of its deadlock variable. The deadlocked processes will have their deadlock variables set to true while all the other processes will have their deadlock variables set to false. If a resource is released, no privileges are generated in the first algorithm because the process will no longer have any outgoing edges. In the second algorithm, a resource release may cause privileges to be created. Some of the variables may be incorrect, but all variables will be up-to-date once the system stabilizes.
2.2 Single Outstanding Request Algorithm

In Detection Algorithm 1, each process has a dep variable. This variable contains the process id of a process which depends on i. When the process first enters the wait-for graph, the process sets its dep variable to its own process id. After this, it will only change the variable to a smaller value according to its direct dependent’s dep variables. Eventually, a process i’s dep variable will contain the lowest id of the processes which depend on i either directly or indirectly. If a process gets the same dep value from two different processes or it gets its own id back, then it knows that it is deadlocked. Once a deadlock is determined by one process, the deadlock variables will be passed around the system to the deadlocked processes. Process i does this by examining the deadlock variable of its successor.
{Detection Algorithm 1 : deadlock detection for process, i}

1. (request not granted) —► dep := P; deadlock := false; source := P

2. (k ∈ PRED) ∧ (dep > dep_k) —► dep := dep_k; source := k

3. (k ∈ PRED) ∧ (dep = dep_k) ∧ (dep = P_i) ∧ (deadlock = false) —►
   deadlock := true

4. (k ∈ PRED) ∧ (dep = dep_k) ∧ (source ≠ k) ∧ (deadlock = false) —►
   deadlock := true; source := k

5. (deadlock = true) ∧ (deadlock = false) —► (deadlock := true)

Lemma 2.1 Detection Algorithm 1 does not detect false deadlocks.

Proof: Since there are only single outstanding requests, the wait-for graph will be split into 0 or more graph structures where each node can have at most one outgoing edge (single outstanding request). So, each graph structure can be examined separately. If there is no deadlock, there is no cycle in the wait-for graph. The only way that a false deadlock would be detected is if either Step 3 or Step 4 were to be executed. In Step 3, if k is a direct dependent of i, dep = dep_k can be true only for a process other than the lowest numbered process (one of its successors). So, P_i ≠ dep and the action in Step 3 will never be executed. In order for source ≠ k (k is a direct dependent), a process must have received the same value from two different processes. But, there is no cycle, and at most one outgoing edge. The only way for source ≠ k to be true is if there is a cycle. So, the action in Step 4 is never executed. If there is no deadlock in the system, deadlock = false for all processes. So, no false deadlocks are detected.

Lemma 2.2 Detection Algorithm 1 detects all deadlocks.

Proof: If the lowest numbered process in the graph structure being examined is in a cycle (see Figure 2.2b), then this value will be passed to its successor (Step 2), and the value will be passed along the cycle until it reaches the lowest numbered process (process 0 in Figure 2.2b). At this point, this process can execute Step 3. Eventually, it must execute this step since there are only a finite number of other
privileges that can exist. Once it makes \( \text{deadlock} = \text{true} \), all deadlock variables in the graph structure will be changed to \text{true} (Step 5). All processes know that they are deadlocked.

If there is a deadlock and the lowest numbered process is outside the cycle, then this lowest value is passed up until it reaches a process that is actually in the cycle. Call this the “first” process in the cycle (in Figure 2.2c, process 1 is the “first” process). Once it reaches this point, the value is passed to its successor and this value travels around the cycle. After it travels around the cycle, it reaches the “first” process in the cycle. Now, for two different predecessors, \( k_1 \) and \( k_2 \), \( \text{dep} = \text{dep}_{k_1} \) and \( \text{dep} = \text{dep}_{k_2} \). So, \( \text{source} \neq k_1 \) or \( \text{source} \neq k_2 \) (in Figure 2.2c, for process 1, \( \text{source} = 0 \), but it also receives 0 from process 3 and \( 3 \neq 0 \)). This process can now and will eventually, execute \( \text{deadlock} = \text{true} \) (Step 4). All the deadlock variables are again passed around the system, and every process will know if it is deadlocked. \( \square \)

**Theorem 2.1** Detection Algorithm 1 is a self-stabilizing deadlock detection algorithm.

*Proof:* There are only a finite number of moves to be made because the process identification numbers are bounded from below. This is why the minimum state in the system is used as the state being passed around. No matter how many processes join the system, eventually the algorithm will terminate because there must be a lower bound on the process identification numbers. This lowest process number in each part of the graph is passed to the successor. Eventually, each process will have its lowest numbered dependent be the value for its \( \text{dep} \) variable. Step 2 can only be executed a finite number of times. Once Step 3 or Step 4 is executed, the deadlock variable is changed to be true and these steps can not be executed any more. Now, the deadlock variables in the entire graph are changed to be true, and there are no more privileges in the system. So, eventually there will be no privileges in the system. Once this occurs, all processes will know whether they are deadlocked or not (by Lemmas 2.1 and 2.2). \( \square \)
2.3 Multiple Outstanding Requests

Detection Algorithm 2 uses successor sets in order to detect deadlocks. Each process maintains its own successor set. The successor set for process $i$ contains the process ids of all processes which $i$ depends on (directly or indirectly). Once there are no privileges in the system, all the successor sets will be up-to-date. If a process finds its own id in its successor set, it knows that it is deadlocked. All processes that are deadlocked will determine the deadlock once their successor sets are complete.

\begin{enumerate}
\item \text{(request not granted)} \implies \text{deadlock} := \text{false}; \text{Succ} := \emptyset
\item \text{Succ} \neq \text{Succ}_1 \cup \text{Succ}_2 \cup \ldots \cup \text{Succ}_k \cup \{P_1, P_2, \ldots, P_k\} \implies \\
\text{Succ} := \text{Succ}_1 \cup \text{Succ}_2 \cup \ldots \cup \text{Succ}_k \cup \{P_1, P_2, \ldots, P_k\}
\item \text{If (P_i \in \text{Succ})} \implies \text{deadlock} := \text{true}
\item \text{(deadlocks = true)} \implies \text{deadlock} := \text{true}
\end{enumerate}

Lemma 2.3 Detection Algorithm 2 does not detect false deadlocks.

\textbf{Proof:} Assume that there is no deadlock in the system. The wait-for graph will not have any cycle. When a process makes a request that is not granted, it sets its \textit{deadlock} variable to \textit{false}. The only way for a process to change its \textit{deadlock} variable to \textit{true} is to execute either Step 3 or Step 4. So, in order for a deadlock to be detected, Step 3 must be executed (the privilege in Step 4 is not true for any process until Step 3 is executed at least once). In order for a process to be able to execute Step 3, its own id must be in the \textit{Succ} set. If this is the case, its id must also appear in the \textit{Succ} set of one of its successors (i.e., Step 2 must have been executed), call this successor $j$. This means that process $i$ is both a predecessor and successor of process $j$ in the wait-for graph, but then there would be a cycle in the graph. This contradicts the original assumption. Thus, the algorithm can not detect a false deadlock. \qed

Lemma 2.4 Detection Algorithm 2 detects all deadlocks.

\textbf{Proof:} Assume there is a deadlock in the system. There must be a cycle in the wait-for graph. Let $P_1, P_2, \ldots, P_k$ be the processes in this cycle (where $P_i$ is a
predecessor of \( P_{i+1}, i = 1, 2, ..., k - 1 \). Eventually, \( P_k \) will appear in the \( \text{Succ} \) set of \( P_{k-1} \) (either it is already in the set, or it executes Step 2). \( P_k \) will then eventually appear in the \( \text{Succ} \) set of \( P_{k-2} \). This can be continued, and \( P_k \) will eventually appear in the \( \text{Succ} \) set of \( P_1 \). Since \( P_1 \) is the successor of \( P_k \), after \( P_k \) executes Step 2, \( P_k \) will be in its set. Now, it will eventually execute Step 3, and the deadlock will be detected. □

**Lemma 2.5** All \( \text{Succ} \) sets will be up-to-date when there are no privileges in the system.

*Proof:* The proof will be by induction. Initially, all \( \text{Succ} \) sets are empty (no requests have been made). Assume that the \( \text{Succ} \) sets are correct. Now, a request or release is made.

Case 1: A request is made by a process, process \( i \). Now, this process has a new successor. The only privilege in the system will be for process \( i \). Process \( i \) will eventually execute Step 2, and its \( \text{Succ} \) set will be correct. The predecessors of \( i \) will eventually execute Step 2, and they will be correct. This continues until there will be no privileges in the system.

Case 2: A resource is released by a process. If no process was waiting for this resource, then no privileges are generated. If this release causes an edge to be deleted in the wait-for graph, the process that was granted the resource no longer has \( i \) as its successor, so it has a privilege. It executes Step 2, and then its predecessors will execute Step 2. This will continue until there are no privileges. At this point, all processes will have correct \( \text{Succ} \) sets. □

**Theorem 2.2** Detection Algorithm 2 is a self-stabilizing deadlock detection algorithm.

*Proof:* By Lemma 2.3, Detection Algorithm 2 does not detect false deadlocks. By Lemma 2.4, if there is a deadlock, a process that is in a cycle will detect a deadlock. By Lemma 2.5, all the successor sets will be correct when there are no privileges in the system. Once they are all correct, any process that is in a cycle will have a
successor set which contains its own process id. Eventually, it will set its deadlock variable to true (Step 3). The processes that are not in any cycles but depend on deadlocked processes will declare deadlock using Step 4. So, all processes that are deadlocked will eventually change their deadlock variable. Each resource request and release causes the Succ sets in the system to be incorrect. This creates privileges in the system, and by Lemma 2.5, eventually there will be no privileges in the system once all the sets are propagated around the system (this will be done in finite time). So, Detection Algorithm 2 is a self-stabilizing deadlock detection algorithm.

2.4 Deadlock Resolution

The algorithms presented in this chapter are self-stabilizing deadlock detection algorithms which use state variables instead of probes to detect a deadlock. Each process knows whether or not it is deadlocked. Once the deadlock is found, the system must resolve the deadlock.

The first thing that must be done is to select a victim process that must be rolled back. This process releases its resources, and the other processes continue to run. This victim must be in the cycle. Otherwise, rolling this process back will not affect the cycle.

In Detection Algorithm 1, there is a process \( i \) that detects the deadlock "first". So, the statement \( \text{victim} = P_i \) can be added to the move in Steps 3 and 4. This victim variable can be passed on to all other processes in the graph so that they can update their status. However, this "first" process may not be the best process to remove. An alternative way would be to replace Step 5 by the following:

\[(5) \ (k \in \text{PRED}) \land (\text{deadlock}_k = \text{true}) \land \text{deadlock} = \text{false} \rightarrow \text{deadlock} = \text{true} \]

If this is done, the deadlock values are passed around the graph, but only those processes in the cycle change their values. After this is done, some method can be used to choose among the processes in the cycle. This is a form of leader election.

In Detection Algorithm 2, if Step 4 is eliminated, then only the processes in
the cycle will detect the deadlock, and a leader election algorithm can be used to
determine the victim.

In Detection Algorithm 1, once the cycle is removed, the other processes need to
update their states because the lowest numbered process may not be in the system
anymore. This can be done by having each process have a complete variable. When
a deadlock is detected, the complete variables are set to false. Once the cycle is
removed, all the processes with complete = false will set their dep variables to their
own process identification number, and then set complete = true. Once this is done,
the number of privileges will eventually be zero, and any request will again trigger
the algorithm. In Detection Algorithm 2, once the process is eliminated from the
cycle, the graph is changed, and the processes still in the graph will automatically
update their Succ sets (Step 2). Since a process has been removed from the wait-for
graph, its predecessors will have incorrect successor sets. These incorrect sets will be
changed, and eventually, these changes will be propagated throughout the system.

2.5 Conclusions

This chapter presents two self-stabilizing deadlock detection algorithms. A single
request model is used in Detection Algorithm 1, but this assumption is not needed in
Detection Algorithm 2. The algorithms use states instead of using probes to detect
deadlocks. This allows the algorithms to update the status dynamically instead of
requiring a process to initiate the algorithm. So, whenever the system is forced into
an illegal state by some resource request or release, the algorithms automatically start
trying to put the system back into a legal state. In finite time, the system will again
be in a legal state. A legal state is defined to be a state in which all processes know
if they are deadlocked, and there are no privileges in the system.

Once a deadlock is detected, there are several ways in which the deadlock can be
resolved. The resolution algorithm can also be constructed similarly to the detection
algorithms.

The two self-stabilizing algorithms have advantages over traditional deadlock de-
tection algorithms. In these traditional algorithms, probes are initiated by processes. So, each process must decide whether or not to send a probe. In the self-stabilizing algorithms, each request automatically causes the processes to change states if they have a privilege. Once the information propagates, all the processes have knowledge of deadlocks and not just the process that sent the probe. This makes resolution simple since all the processes know whether the deadlock is affecting them or not.
Chapter 3

DEADLOCK PREVENTION

One way of preventing deadlocks is to continually rank the nodes to determine if an edge that is added will create a cycle in the graph [7]. If a cycle will occur when the edge is added in the wait-for graph, the edge is not granted. In other words, if a potential deadlock is detected then the resource allocation is not done, and this process is rolled back. So, the deadlock is prevented. This is very expensive because the algorithm is run each time a request can not be satisfied.

For a given process $i$, the local state of the process can be read by $i$ and all of its neighbors (successors of $i$ for the algorithm discussed here). According to the states of its neighbors, a process can change its own state. The local states define a global state of the system which is the cross product of these local states. The global states can be split up into legal and illegal configurations.

This chapter presents a self-stabilizing deadlock prevention algorithm. Instead of using a ranking system, each process in the system has a certain state, and a process can determine whether or not it is in a potential deadlock by examining this state. The legal global states in the system are defined by the knowledge of possible deadlocks. If the system is in a legal state, all the processes that may be deadlocked should know the effects of adding the edge (whether the addition of the edge will create a cycle or not). Every resource request that is not granted and every release
of a resource causes the system to be put into an illegal state. Once a process knows that the addition of the edge will cause a cycle, it releases all its resources. This is complete rollback. It is safer than just not granting the request edge (adding the edge), but the algorithm can be modified to make only a partial rollback.

3.1 Variables

The algorithm uses successor sets to determine a potential deadlock in the system. Each process maintains a set called the successor set. The successor set of process $i$, $Succ_i$, is defined as follows: $\forall j \in SUCC_i, Succ_j \subseteq Succ_i$ where $SUCC_i$ denotes the immediate successors of $i$ in the wait-for graph. Each process, $i$, maintains its own successor set, $Succ_i$. If a process $i$ has $k$ immediate successors, then $Succ_1, Succ_2, \ldots, Succ_k$ are the successor sets for all successors of $i$. Along with the successor sets, each process, $i$, also has a local successor set, $localS_i$. This set contains only the direct successors of $i$ (when the system is in a legal state). If $j \in localS_i$, then there exists an edge from $i$ to $j$ in the wait-for graph. These local sets help propagate correct information around the system.

3.2 Deadlock Prevention Algorithm

This algorithm uses successor sets in order to detect potential deadlocks. For process $i$, $Succ_i$ contains the process ids of all processes which $i$ depends on (directly or indirectly). Once the system stabilizes, the successor sets may not give the exact information of the dependencies (there may be extra ids in the sets due to faulty initialization), but they will be able to tell whether or not there is a cycle. If a process, $i$, finds $i \in Succ_i$, it knows that it is in a cycle. Once a potential deadlock is detected, a process in the cycle will release its resources. The cycle (corresponding to the potential deadlock) will be removed.

The local sets are used so that the proper information is eventually passed around
the system. Eventually, the local sets will be correct \((\forall i, localS_i = SUCC_i)\). After this, the information is passed around. If a processor is not in a cycle and not dependent on a process in a cycle, the \(SUCC_i\) set will be correct and reflect all the successors of the processor. If a processor is in a cycle or depends on a process in a cycle, the \(SUCC_i\) sets may or may not reflect the correct successors. However, once the system stabilizes, the processors in the cycle will have their own id in the successor set.

{Deadlock Prevention Algorithm for process \(i\) which has \(k\) successors.}

1. \((localS_i \neq SUCC_i) \rightarrow localS_i := SUCC_i\)
2. \((localS_i = SUCC_i) \land (SUCC_i \neq localS_i \cup \bigcup_{n \in SUCC} (localS_n \cup Succ_n)) \rightarrow\)
   \[SUCC_i := localS_i \cup \bigcup_{n \in SUCC} (localS_n \cup Succ_n)\]
3. \((localS_i = SUCC_i) \land (i \in Succ_i) \rightarrow release resources; Succ_i := Succ_i - \{i\}\)

Statement 1 forces the local successor sets to converge to the correct values. Statement 2 is used to build the successor sets. Statement 3 is the prevention statement. Once a process finds its id in its successor set, it releases its resources (there are other modifications that can be done for this step which are discussed in Section 3.4). For the system to stabilize, there must be a time period where no perturbances occur (resource requests or releases). If perturbances occur continually so that the system never stabilizes, deadlocks will still be prevented because cycles will still be detected and removed, but the successor sets will not stabilize. The lemmas and theorem assume that there will be no requests or releases (except those initiated because of the algorithm) in order for the system to stabilize.

**Lemma 3.1** Eventually, all local successor sets will be correct.

**Proof:** It is enough to show that given a process \(i\), it will eventually have a correct local successor set. If \(i\) ever executes the action in Statement 1, \(localS_i\) will be correct. All that needs to be shown is that \(i\) will eventually execute the action in Statement 1. If the local successor set is not correct, then \(i\) can not have a privilege through statements 2 or 3. Eventually, \(i\) will execute the action in Statement 1. After this is done, \(i\) will have a correct local successor set. \(\square\)
Lemma 3.2 If no perturbances occur, all Succ_i sets will stabilize.

Proof: By Lemma 3.1, all local successor sets will be correct. Once this occurs, this information is propagated around the system. In order for the Succ_i sets to continually change, the action in Statement 2 must be executed by some process. If there is no cycle in the system, then the processes with no successors will eventually have stable successor sets. Every other process in the system can have successors, but if the successors are traced, they all end at a process with no successors (process i can only change if one of its successors changes, this successor can only change if one of its successors changes, and so on until a node with no successors is reached which can not change its successor set). Because of this, the stable "end" nodes will cause all other successor sets to stabilize. If there is a cycle, all the processes in the cycle will eventually have the same successor set. Any change in one of them will be propagated around the cycle. The ids in the successor set will be the ids of the processes in the cycle along with all of the ids which processes in the cycle depend on (directly or indirectly).

Corollary 1 Once the successor sets stabilize, i ∈ Succ_i if i is in a cycle.

Corollary 2 Once the successor sets stabilize, i ∉ Succ_i if i does not depend on a process that is in a cycle.

Theorem 3.1 The algorithm is a self-stabilizing deadlock prevention.

Proof: By Lemma 3.2, all successor sets will stabilize. If there is a cycle and no releases are made which break the cycle, each process in the cycle will eventually have its own id in its successor set (Corollary 1). After this occurs, a process in the cycle must execute the action in Statement 3. This will continue to occur until eventually the cycle is eliminated. So, any cycle in the graph will be removed. If there are no cycles in the wait-for graph, eventually all successor sets will stabilize and no process will have its own id in its successor set (Corollary 2). At this point the system is stabilized. Once stabilization occurs, there are no deadlocks in the system. So, the algorithm is a self-stabilizing deadlock prevention algorithm.

□
### 3.3 Example of Algorithm

Figure 3.3a shows an example wait-for graph for a system with 7 processes when process 7 requests a resource held by process 2. Once the edge is created, privileges are created in the system. Table 1 shows the successor sets for each processor at each step. For simplicity, all the local successor sets are assumed to be correct. However, this assumption is not needed for the algorithm to operate correctly.

Figure 3.3 shows the successor sets when they are stabilized. At this point, process 7 knows that it is deadlocked. It is the first to realize it, and it will release its resource to process 6. The resulting wait-for graph is shown in Figure 3.3b. Once this occurs, the change is propagated back through the system. Figure 3.3 shows the resulting successor sets at each step. The deadlock is prevented, and the system is stable until a perturbation occurs.
3.4 Modifications

The prevention statement in the algorithm can be modified for better efficiency. Once a process determines that there is a possibility of a deadlock, it releases its resources. The potential deadlock is removed, but before this knowledge is passed around the system through the successor sets, another process may unnecessarily release its resources. This may or may not be desired. There is a way to combat this problem. Instead of immediately releasing resources, the process can wait a certain amount of time before releasing its resources. This will allow the changes to propagate around the system. Statement 3 would also be modified:

\[(3) \text{ (local} S_i = SUCC_i) \land (SUCC_i = local S_i \cup \bigcup_{n \in SUCC_i} (local S_n \cup SUCC_n)) \land (i \in SUCC_i) \rightarrow \text{release resources; } SUCC_i := SUCC_i - \{i\}\]

This extra check will decrease the number of processes that release its resources. Also, if the cycle has only a length of two and one process releases its resources, then the other process will not release its resources.

This change does not solve the problem if the system requires that only one process releases its resources. In order to solve this problem, an exceptional process can be used. The process that caused the most recent edge to be created could be considered the exceptional process. This process runs the algorithm while the other processes run the same algorithm without Statement 3.

This would mean that all the other processes in the system would just update
their successor sets. The only process that would detect the potential deadlock is the process that caused the most recent edge in the graph to be added. This process will be the process that releases its resources and breaks up the cycle. However, this will only work if the system stabilizes before another change to the wait-for graph is allowed.

In order to change the algorithm from the total rollback to only partial rollback, the exceptional machine is again used. But, once this process detects a cycle, instead of releasing all resources, the request for the edge is simply not granted. Another way is to release only those resources which process $i$ thinks will break up the cycle. This can be done by examining the predecessors successor sets. If a predecessor, $k$, has $k \in \text{Succ}_k$ then that resource should be released (the edge from $k$ to $i$ will be removed) since that process is probably in the same cycle.

Exceptional machines are used in many self-stabilizing algorithms [4], and this exceptional machine fixes the problem of several processes releasing their resources at the same time.

### 3.5 Conclusions

This chapter presents self-stabilizing deadlock prevention algorithm. The algorithm uses machine states instead of probes [3, 8, 9] or a ranking system [7] to prevent deadlocks. This allows the algorithm to be self-stabilizing. So, whenever the system is forced into an illegal state by some resource request or release, the algorithm automatically starts trying to put the system back into a legal state. In a finite amount of time, the system will again be in a legal state. A legal state is defined to be a state in which there is no cycle in the wait-for graph, and there are no privileges in the system. Once a deadlock is suspected by a process, a process in the cycle will release its resources which prevents the deadlock from occurring.

Section 3.4 discussed possible modifications to the algorithm in order to prevent several processes from releasing their resources when a potential deadlock is detected. The modification that forces only one process in the cycle to release its resources uses
an exceptional machine (the process that made the most recent request) in order to have only this exceptional process release its resources.

A self-stabilizing prevention algorithm has advantages over deadlock prevention algorithms which are not self-stabilizing. In these other algorithms, probes are initiated by processes. So, each process must decide whether or not to send a probe. But, in the algorithm presented here, each request automatically causes the processes to change states if they have a privilege. Once the system stabilizes, the deadlock is automatically prevented. So, this algorithm automatically detects potential deadlocks and prevents them in a simple manner. So, the prevention of deadlocks does not have to be initiated by any process. the prevention is automatically done.
Chapter 4

CONCLUSIONS

Self-stabilization is an evolving paradigm in fault-tolerant computing. Dijkstra originally introduced the property of self-stabilization in distributed systems by developing three self-stabilizing mutual exclusion algorithms. These algorithms were all non-uniform algorithms because an exceptional machine is used. The algorithms were also shown to be correct only in the presence of a central demon (however, they also work in the presence of a distributed demon).

After this paper, self-stabilizing mutual exclusion was researched a great deal. The areas of focus included, number of states required, uniform versus non-uniform versus symmetric, type of demon required, etc. Now, the study of self-stabilization has started to expand to other areas of distributed systems [6]. This research focused on an area to which no one has tried to apply self-stabilization, deadlock detection and prevention.

Chapter 2 discusses two self-stabilizing detection algorithms. The first algorithm (Detection Algorithm 1) used the single outstanding request model. The algorithm is used as a building block for the second algorithm, Detection Algorithm 2. In this algorithm, processes can have any number of outstanding resource requests. In both algorithms, all processes will know whether they are deadlocked are not when the system stabilizes. Once this happens, the deadlock can be resolved. Methods of
resolution are discussed in Section 2.4.

Chapter 3 discusses a method of preventing deadlocks in a self-stabilizing manner, Prevention Algorithm. Once the system stabilized, the system is guaranteed to be deadlock free. The general multiple outstanding request model is used for this algorithm as well. Enhancements can be made to this algorithm depending on the requirements of the system. Some of these enhancements are presented in Section 3.4.

There are several reasons why self-stabilizing deadlock algorithms are better than traditional deadlock algorithms:

- The algorithm runs continually (no initiation of the algorithm needs to be done).
- Any resource request or release automatically creates privileges in the system. Once the system stabilizes, there will be no privileges (any deadlock will be detected or prevented).
- No initialization of the local variables needs to be done, because a self-stabilizing algorithm does not require any initialization.
- The statements in the algorithm can be executed in any order, and the system will still stabilize.
- The algorithm automatically tolerates transient errors (message loss, message corruption, etc.).

These reasons along with the simplicity of the algorithms makes all three algorithms easier to implement and more fault-tolerant than traditional deadlock detection and prevention algorithms.
Bibliography


