Design of deadlock detection and prevention algorithms in distributed systems

Ramesh Dutt Javagal

University of Nevada, Las Vegas

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

Repository Citation
https://digitalscholarship.unlv.edu/rtds/171

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.
Design of deadlock detection and prevention algorithms in distributed systems

Javagal, Ramesh Dutt, M.S.
University of Nevada, Las Vegas, 1992
DESIGN OF DEADLOCK DETECTION AND
PREVENTION ALGORITHMS IN DISTRIBUTED
SYSTEMS

by

Ramesh Dutt Javagal

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, 1992
The thesis of Ramesh Dutt Javagal 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, 1992
ABSTRACT

A distributed system consists of a collection of processes which communicate with each other exclusively through messages to achieve a common goal. These processes may run concurrently on separate physical processors which are not connected to any global memory. Local states of the processes are maintained in memories local to the processors running the processes. In absence of a shared memory, processes communicate through messages. The communication is asynchronous, and a message may take an arbitrary but finite amount of time to move from one process to another. One main problem in distributed systems is the possibility of deadlock. Processes are said to be deadlocked when some processes are blocked on resource requests that can never be satisfied unless drastic systems action is taken. This research work contributes in two approaches of deadlock - detection and prevention.

Two distributed deadlock detection algorithms handling multiple outstanding requests is proposed. The algorithms are proven to be correct: it detects all cycles and does not detect false deadlocks. Also, some simulation results, comparing one of the proposed algorithms with some existing algorithms is also presented. Results of simulation show that the proposed algorithm performs very well in the number of messages required for detecting a cycle.

A new method of preventing deadlocks in resource sharing is proposed. The algorithm is based on the notion of coloring the nodes of the waitfor graph. Rollback is quite less compared to some existing algorithms.
ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Ajoy Kumar Datta for inspiring me throughout the course of this thesis. His constant encouragement and stimulating environment made me work more than my capacity. I would like to thank Dr. Thomas Nartker and Dr. John Minor for their support throughout my education and research at UNLV. I would like to thank Army Research Office for providing me with funds to pursue my advanced education and research. This research was supported in part by the U. S. Army Research Office under grant DAAL03-87-G-0004. Without their support this thesis would not have been possible. Special thanks to Dr. Kazem Taghva, Dr. Laxmi Gewali, and Dr. Ashok Iyer for serving on my committee and for their helpful comments. I would also like to thank the UNLV College of Engineering systems staff for helping me out understanding many aspects of the system features. Special thanks to John D. Kilburg and Brian Johnston for the very interesting and enlightening discussion on various parts of my thesis. I owe a general debt to other friends in the College of Engineering for their friendship, harassment, and occasional help.

This thesis is dedicated to my family. I would like to thank my parents for sacrificing so much for me; my brother, his wife, and my sister for their inspiration, encouragement, and support. Special thanks are due to my wife, Nagarathna R. Dutt for providing a stable and supportive working environment throughout. Most of all, I thank God for giving me the wisdom and patience required to complete this work and the degree program.
Contents

Abstract ............................................................................................................................... iii
Acknowledgments ............................................................................................................... iv

1 INTRODUCTION ............................................................................................................ 1

2 DEADLOCK DETECTION ............................................................................................. 5
  2.1 Related Work ........................................................................................................... 5
  2.2 Algorithm 2.2 ......................................................................................................... 6
     2.2.1 Basic Concepts .............................................................................................. 7
     2.2.2 The Waitfor Graph ...................................................................................... 7
     2.2.3 Building a signalling mechanism .................................................................. 8
     2.2.4 More about Signals ...................................................................................... 11
     2.2.5 Signal Handling ......................................................................................... 13
     2.2.6 Resolution ................................................................................................. 15
     2.2.7 Proof of Correctness ................................................................................. 16
     2.2.8 Example ...................................................................................................... 17
  2.3 Algorithm 2.3 ......................................................................................................... 19
     2.3.1 Model Description ...................................................................................... 19
     2.3.2 Explanation of the Algorithm ..................................................................... 20
     2.3.3 Proof of Correctness .................................................................................. 22
     2.3.4 An Example .............................................................................................. 23
  2.4 Simulation .............................................................................................................. 23
     2.4.1 Method ....................................................................................................... 26
## List of Figures

2.1 A sample waitfor graph. ................................................................. 7

2.2 The building of a waitfor graph. (a) $p_4$ requests for a resource owned by $p_3$.
(b) This resource is currently being used by $p_7$, and $p_5$ is the last process in
the queue for this resource. (c) $p_5$ eventually sends an ack[request] to $p_4$. ... 9

2.3 Avoidance of deadlock in the signalling mechanism. ....................... 10

2.4 The priority structure amongst the signals. ........................................ 12

2.5 Distributed system with five processes. .......................................... 17

2.6 Occurrence of a deadlock. ............................................................. 18

2.7 Resolving the deadlock by aborting $p_1$ from the system. ................ 18

2.8 Transactions and Data Objects in Site $S_1$. ........................................ 19

2.9 A distributed database system with seven transactions. .................... 24

2.10 The system after cycles are formed. ............................................. 25

3.1 The building of a waitfor graph. (a) $p_4$ requests for a resource owned by $p_3$.
(b) This resource is currently being used by $p_7$ and $p_5$ is the last process in
the queue for this resource. (c) $p_5$ sends an ack to $p_4$. ....................... 38

3.2 Example of a process rollback. (a) process $p_1$ makes a request for a resource
to process $p_3$. (b) process $p_1$ rolls back. ............................................ 39

3.3 A waitfor graph with each node labeled by its present color. ................. 41

3.4 Distributed system with 10 processes. ............................................. 46

3.5 The status of the variables maintained at each node. ....................... 47

3.6 Example of an unnecessary rollback. $p_6$ sends a request to $p_9$. $p_9$ sends a
nack and $p_6$ rolls back. ............................................................ 47
3.7 Creation of a cycle is avoided. $p_9$ sends a request to $p_8$. $p_8$ sends a nack to $p_9$ and $p_9$ rolls back (unavoidable rollback).

3.8 Inactive node processing a request. $p_3$ sends a request to $p_6$. $p_6$ is inactive and sends an ack.

3.9 Active node processing a request for a resource. $p_{10}$ sends a request to $p_4$. $p_4$ sends an ack. $p_3$ sends a request to $p_6$. $p_6$ is active since $p_{10}$ has not sent the deactivate signal yet. $p_6$ sends a nack to $p_3$ and $p_3$ rolls back (unnecessary rollback).

3.10 The situation after $p_3$ rolls back.

3.11 An example to show starvation. Process $p_3$ starves.

3.12 Preemption of a nonterminal node.
Chapter 1

INTRODUCTION

A distributed computer system consists of a set of autonomous processes linked by a network. Processes in such an environment do not have any global memory, but communicate through messages. 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. The processes may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. The four major reasons for building distributed systems are: resource sharing, computation speedup, reliability, and communication.

The processes on these sites can use resources or share information local to them or available over the network. The processes may request for resources in any order which is not known a priori. The requested resources may be available or locked by other processes, thus building a graph called wait-for graph. In general, resource sharing in a distributed system provides mechanisms for sharing files (resources) at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices and other operations.

If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then the availability of a distributed system may allow us to distribute the computation among the various sites, to run it concurrently. If a particular site is overloaded, some of the jobs may be moved to other lightly loaded processes. If a system
is composed of a number of large autonomous installations, the failure of one of them should not affect the rest. The failure of a processor can be detected by the system, and appropriate action may be needed to recover from the failure. When the failed site recovers, or is repaired, mechanisms must be available to integrate it back into the system smoothly.

The advantages of a distributed system are: predictable response, cost, extensibility, and availability and reliability, while the disadvantages are loss of flexibility in the allocation of memory and processing resources, dependence on network performance and reliability, and security weaknesses.

Deadlock

Processes are said to be deadlocked when some processes are blocked on resource requests that can never be satisfied unless drastic systems action is taken. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). A computer can normally have many different resources that can be acquired. Some resources may be available in several identical instances, such as three tape drives, etc. Any one of such instances can be used to satisfy any request for the resource. Deadlock has become one of the main problems in the field of distributed systems as the set of the running processes might request for the same resources and no single process can start executing as each process is waiting on another process which is a part of this cycle. In other words, once a deadlock occurs, the set of processes involved in the cycle will never do any useful computation, unless the deadlock is broken by some action.

A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. Each resource is either correctly assigned to exactly one process or is available.

2. Hold and wait. Processes currently holding resources granted earlier can request new resources.

3. No preemption. Resources previously granted cannot be forcibly taken away from a
4. Circular wait. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

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 currently held it. The advantages of deadlock detection routines are that once the routines to detect deadlock are developed, they can be used with any arbitrary system, while the disadvantage is the run-time overhead. Detecting the deadlock in distributed environment is very hard and selection of the victim process can have important repercussions for system performance. However, the selection of the process to be aborted is highly system-dependent and hence it seems fruitful in finding a correct, low-overhead deadlock detection scheme. The technique adopted in detecting a deadlock is based on sending messages along the edges of the wait-for graph.

A number of algorithms have been proposed for detecting deadlocks in distributed systems [2, 3, 29, 32, 36]. 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)[32], 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.

Some of the algorithms in the literature have been found to detect deadlocks which do not exist. This situation is generally termed as false deadlocks. The disadvantage of detecting false deadlocks is that processes are aborted unnecessarily, thus decreasing the system performance. As the processes need to send messages across sites, the message overhead increases, so is the increase in storage requirements, as processes have to store wait-for information.

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 to be found, 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. Bernstein, et al.[1] give theoretical reasons for the predominance of short paths in wait-for graphs.

In deadlock prevention, the system is designed such that a 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 deadlock [18]. The methods adopted in [18, 34] to pre-allocate all the requested resources no longer are 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 be really deadlock-free, as possible deadlocks can be easily overlooked when reasoning informally about a system. Timestamp 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.

In deadlock avoidance, some knowledge of the future process behavior is used to constrain the resource allocation to avoid deadlock in the system. Various schemes have been proposed to avoid deadlock in a centralized scheme[20, 30]. Srimani, et. al.[37] have proposed a new heuristic algorithm to solve the problem of deadlock avoidance in a distributed system with multiple resource types.

In this research we are concerned only with deadlock detection and prevention in distributed systems.
Chapter 2

DEADLOCK DETECTION

This chapter deals with methods of detecting deadlocks in resource sharing for distributed systems. The algorithms are based on sending messages along the edges of the waitfor graph. Algorithm 2.2[21, 22] is built on a prioritized signalling mechanism which can be implemented on an underlying routing protocol, while Algorithm 2.3[23, 24] uses an update message instead of probe messages. The algorithms support multiple resources and multiple outstanding requests. The proposed algorithms avoid the detection of false deadlocks, and are capable of detecting deadlocks involving a subset of processes in the system. The algorithms work well even when multiple nodes initiate the deadlock detection algorithm. An informal argument of the proof of correctness of the proposed algorithms are also presented. A comparison of the algorithms with other existing distributed deadlock detection algorithms is also briefly presented.

2.1 Related Work

In such a resource sharing environment, deadlock is a potential danger. When a set of processes enter into such a state that each process waits for some other process in this set to release a resource, all the processes are blocked indefinitely. In a shared memory environment, a number of deadlock detection algorithms are available [10, 26, 35]. However, in a distributed environment, the added complexity of these algorithms is due to the
unpredictable propagation delays, and the consequent nonavailability of the global state. With the exception of [2, 3, 4, 5, 6, 15, 19, 36, 38] most of the other known deadlock detection algorithms [29, 32] first undergo a state collection process, and then detect possible cycles in the waitfor graph. Due to inconsistency in the collected global state, many of these algorithms are prone to false deadlocks[13]. The proposed algorithms utilize some properties of the waitfor graph to minimize the state collection procedure, and embeds a signalling mechanism to overcome false deadlocks. A new approach to detect deadlocks using the concept of self-stabilization [8, 12] is reported in [11]. Kshemkalyani, et. al.[27, 28] attempts a formal proof of the correctness of a deadlock detection/resolution algorithm and suggests that invariant-based techniques can be used to prove the correctness of distributed algorithms.

Elmagarmid [10] shows that the proof of correctness of the algorithm by Obermarck[32] is incorrect with the following observation: The portions of the wait-for graph that are shipped around may not represent a consistent view of the global wait-for graph, since each site takes its snapshot asynchronously. Knapp[26] shows that the algorithm by Chandy and Misra[2] is incorrect, by providing a counterexample.

2.2 Algorithm 2.2

In this universe of processes forming a distributed system, each component process has a set of local resources owned and managed by it. A process however uses resources which may or may not be local to it. A remote resource can be accessed by sending an explicit request to the owner of that resource. Such resources are often not shareable - exclusive access becomes necessary either due to hardware constraints, or due to software constraints like consistency and determinacy. When one process needs a resource currently being used by another process, it sends a request and waits for that resource to be released. It is assumed that every process is well-behaved in as much as once it acquires a resource, it releases it within a finite amount of time. It is fair to assume that no process has a prior knowledge of the future resource requirements of any process, including itself, in the total system. In the proposed algorithm all true deadlocks are detected and no false (phantom) deadlocks
2.2.1 Basic Concepts

In this algorithm, a process is permitted to request a set of resources. A process can execute only after it acquires all resources it has requested for [2, 3, 13, 29, 32]. As defined in [26], this is an AND model of deadlock, which is strictly more general than the one-resource model. Also the proposed algorithm belongs to the class of edge-chasing algorithms, since the signals are propagated along the edges of the waitfor graph. Messages sent from process A to process B are received by process B in the same order as they were sent.

2.2.2 The Waitfor Graph

Let the distributed system be composed of a set of n processes \( \{p_1, p_2, p_3, \ldots, p_n\} \) which are expected to share the resources in such a way that deadlock does not occur. A waitfor graph is a graph which represents which process is waiting for which other process for the purpose of acquiring a resource. A directed arc from some process \( p_1 \) to another process \( p_3 \) (Figure 2.1) would thus mean that \( p_3 \) is using (or waiting for) some resource which is also required by \( p_1 \), and \( p_1 \) can use it only after \( p_3 \) releases that resource.

Observation 1: A node in the waitfor graph can have more than one outgoing edge and more than one incoming edge.

As a process can make multiple resource requests, a node can have more than one
outgoing edge. In some cases (discussed later), a node can have more than one incoming edge. In the WFG, there could be many initial nodes (with zero incoming edges) and many terminal nodes (with zero outgoing edges).

**Definition 1** In the waitfor graph, a node with zero outgoing edges will be called a terminal node, and a node with zero incoming edges will be called an initial node.

**Lemma 2.1** A node in the waitfor graph can have more than one outgoing edge and more than one incoming edge.

Terminal nodes are said to be executing at the present moment and any node is eligible to make a request for a resource.

Figure 2.1 shows a sample waitfor graph. $p_1, p_2$ are the initial nodes and $p_4, p_5$ are the terminal nodes.

### 2.2.3 Building a signalling mechanism

Before going into the details of the signalling mechanism, we define successor-set which has important implications in the paper.

**Definition 2** A successor-set of a process contains all the owner processes of the resources it has requested.

When a process $p_i$ needs a resource owned by another process $p_j$, it sends a request to $p_j$, and sets up a provisional arc from $p_i$ to $p_j$ (Figure 2.2). If the resource is available, then the request is granted to $p_i$, and the provisional arc is removed. If on the other hand, the resource is not available, then a message is forwarded to the last process $p_k$ which requested for that resource, and the provisional arc from $p_i$ to $p_j$ is converted to a provisional arc from $p_i$ to $p_k$. The process $p_k$ eventually acknowledges this request which is communicated to $p_i$ by sending an ack (acknowledgment) signal via $p_j$, and the provisional arc from $p_i$ to $p_k$ is converted to a true arc from $p_i$ to $p_k$. The owner of the resource $p_j$ is now included in $p_i$'s successor-set. We assume that when a signal is received by a process, the receiving process knows the identity of the sending process.
Figure 2.2: The building of a waitfor graph. (a) $p_4$ requests for a resource owned by $p_3$. (b) This resource is currently being used by $p_7$, and $p_5$ is the last process in the queue for this resource. (c) $p_5$ eventually sends an ack[request] to $p_4$.

The communication of the request and the acknowledgment signals between a pair of processes is thus channelized through the owner of the concerned resource. For every resource which a process owns, the process maintains a resource-queue of the processes waiting for that resource. The waiting process maintains the successor-set containing the processes it has requested for the resources. Once a resource is granted, the owner of that resource is deleted from the successor-set. When a process completes using a resource, it sends a release signal to the owner of that resource, which acknowledges it, and reallocates that resource to the next waiting process by sending a grant signal. This results in the removal of an arc in the waitfor graph. A release signal is also sent when a process is preempted due to any reason. One of the ways a process can be preempted is by rollback.

The pair of signals request, ack[request] constitutes an atomic action in as much as every process receiving a request defers the decision about a subsequent request until it sends the
corresponding acknowledgment signal. However, this oversimplified signalling mechanism itself is prone to deadlock. To break a possible deadlock in the signalling mechanism, we use the process numbers associated with the nodes. As in a priority interrupt scheme, requests with a higher process number would get a higher priority over requests with a lower process number. A typical situation is illustrated in Figure 2.3. Here, process $p_2$ is trying to acquire a resource owned by $p_3$, and $p_3$ is trying to acquire a resource held by $p_1$ with no other processes currently competing for these resources. However, since the process number contained in the request from $p_3$ is higher than the process number of $p_2$, $p_2$ is committed to accept and wait for $p_3$ to receive an $ack$/$request$ before $p_3$ can process and send an acknowledgment to $p_2$.

**Definition 3** An initiator-set of a process $p_i$ is a set of tuples $(p_j, p_k)$, where $p_j$ is the initiator and $p_k$ is the process from which $p_i$ received the find_deadlock message. Once process $p_i$ receives $ack$/$false$(p_j)$ from all its successors, then all tuples containing $p_j$ as the initiator are deleted from $p_i$'s initiator-set.

In the present scheme, to detect a possible deadlock in the system, a process $p_i$ would send a $find$/$deadlock$(p_i, p_i) signal to all of its successor nodes. A $find$/$deadlock$ signal is initiated after $p_i$ has waited for a resource for at least time $T[14]$. The first parameter of the $find$/$deadlock$ signal denotes the process initiating the signal, while the second parameter denotes the process sending the signal. If the successor node $p_j$ is a terminal node, it returns an $ack$/$false$(p_i)$ to the sender process down the edge of the waitfor graph, which implies

![Figure 2.3: Avoidance of deadlock in the signalling mechanism.](image-url)
that there is no deadlock. If however, the successor node, \( p_j \) is not a terminal node, then it forwards the \( \text{find.deadlock}(p_i, p_j) \) signal to its successors after updating the sending process value (i.e., changing the second parameter of the \( \text{find.deadlock} \) signal. The forwarding process stores the initiator of this \( \text{find.deadlock} \) signal (in this case, \( p_i \)) in its initiator-set. An acknowledgment is sent to the predecessor node only after an acknowledgment is received from all of its successor nodes. An \( \text{ack}[\neg \text{false}(p_i)] \) would indicate the absence of a deadlock. Once a node detects a deadlock, it broadcasts to all the processes in the tree of the waitfor graph.

While detecting the absence of deadlocks is fairly straightforward, detecting the presence of deadlocks poses a termination problem, since the waitfor graph becomes cyclic. The paper \cite{9} provides a simple solution to overcome this problem.

**Lemma 2.2** The \( \text{find.deadlock}(p_j, p_k) \) signal received by \( p_i \) is forwarded to the successors of \( p_i \) at most once.

The above approach makes sure that a cycle involving the initiator also is detected, but a cycle not involving the initiator is never detected. When process \( p_i \) receives \( \text{find.deadlock}(p_j, p_m) \), \( p_i \) has no way of determining whether it is a cycle or not. Hence the cycle not involving the initiator goes undetected. This appears to have been the problem in the case of \[3, 5\] also.

### 2.2.4 More about Signals

The set of signals introduced so far can be summarized as follows:

1. \( \text{request}, \text{ack}[\text{request}] \)
2. \( \text{find.deadlock}, \text{ack}[\neg \text{false}] \)
3. \( \text{release}, \text{ack}[\text{release}], \text{grant} \).

These signals are *supervisory signals* and are distinct from the usual interprocess communication signals, which are an integral part of the underlying computation. Any undesirable interaction amongst these supervisory signals can be conveniently overcome by defining a *priority structure* in accordance with their numerical ordering in the list — where \( \text{request} \) has the lowest priority, and \( \text{release} \) and \( \text{grant} \) has the highest priority (Figure 2.4). To
ensure the absence of deadlock in the signalling mechanism itself, it is important to ensure that for every signal $s$ sent or forwarded by a node, the corresponding $\text{ack}[s]$ is eventually received. Since a signal with a higher priority can interrupt the atomic action initiated by another signal with a lower priority, the indefinite blocking is avoided with signals of unequal priority levels. However, since the waitfor graph is a dynamic graph, an additional complication is possible which needs careful attention.

Consider that a node $p_1$ has received and forwarded a $\text{find deadlock}$ signal to its successor $p_2$ and is waiting for the acknowledgment. Meanwhile due to some reason, the process $p_2$ has been aborted, causing the removal of the arc from $p_1$ to $p_2$ in the waitfor graph. In the absence of this edge, the acknowledgment signal cannot return to $p_1$ leading to a deadlock in the signalling mechanism itself!

To overcome such problems, the following approach seems feasible:

**Definition 4** A node in the waitfor graph will be called a pending node, if it has initiated or forwarded a signal but has not yet received the corresponding acknowledgment.

**Lemma 2.3** If a pending node $p_k$ which has sent or forwarded a $\text{find deadlock}(p_j, p_k)$ signal becomes a terminal node, then it should return an $\text{ack}[\text{false}(p_j)]$ to its predecessor. If $p_k$

---

1 The grant signal is of type acknowledgment.
2 Note that release has the highest priority for obvious reasons.
becomes an initial node, then it should discard all the subsequent acknowledgments received by it.

With this modification, there cannot be a deadlock involving signals of unequal priorities. There is no apprehension of deadlock involving a number of find.deadlock signals originating from different processes, since all such signals would flow in the same direction following the directed edges of the waitfor graph. Similar observations are valid for the release signals also. Thus, the signalling mechanism itself is free from deadlock.

2.2.5 Signal Handling

The life of a process in this universe of processes can be described as follows:

* \([\text{compute}]
\quad [\text{send request; receive } \text{ack}[\text{request}] \text{ or grant }] \quad[\text{receive request; handle request; send } \text{ack}[\text{request}]]
\quad [\text{send release; receive } \text{ack[release]}]
\quad [\text{receive release; handle release; send } \text{ack[release]}]
\quad [\text{send find.deadlock}^3 \text{; receive } \text{ack[false]}
\quad [\text{receive find.deadlock; process find.deadlock; send } \text{ack[false] or broadcast deadlock}]
\)

The algorithm for handling the signals request, release, grant and find.deadlock are outlined below (using CSP notation[17]):

```csp
var
successor-set, initiator-set : set of processes;
r : resource;
q : array [1..r] of queue;
um : array [1..r] of integer;
terminal : boolean;
pending : boolean:
free, owner : array [1..r] of boolean;
{terminal is true only when the node is a terminal node;
pending is true only when the node has sent a find.deadlock signal and is waiting for 
ack[false];}
initially, all the nodes are terminal nodes;
q is empty for every resource; pending = false;
}

{process \(p_k\) receiving a request for a resource \(r\) from \(p_j\)}

\(p_j ? \text{request}(r) \rightarrow \)
```

3 The find.deadlock signal is sent only by a process waiting for a resource.
{This is received by the owner of r only}
\[
\begin{align*}
\text{free}[r] & \rightarrow \\
\text{free}[r] & := \text{false}; \\
p_j & \text{ grant}(r) \\
\end{align*}
\]

\![\neg \text{free}[r] \rightarrow \\
\text{[tail.q}[r]! \text{ request}(r); \\
\text{tail.q}[r]? \text{ ack[request}(r)); \\
\text{enqueue } (p_j, q[r]); \\
p_j! \text{ ack[request}(r))}
\]

\[
\begin{align*}
\text{Pj} & \text{ grant}(r) \\
\text{enqueue } (p_j, q[r]); \\
p_j & \text{ ack[request}(r)) \\
\end{align*}
\]

\[
\begin{align*}
\text{process } p_j \text{ receiving an ack[request}(r)] \text{ from } p_k \\
p_k & \text{ ack[request}(r))] \\
\text{[insert } p_k \text{ in } p_j\text{'s successor-set; } \\
\text{terminal} & := \text{false; } \\
\end{align*}
\]

\[
\begin{align*}
\text{process } p_k \text{ receiving a release of a resource r from } p_j \\
p_j & \text{ release}(r) \\
\{ \text{This is received by the owner of r only} \} \\
\text{[dequeue}(p_j, q[r]); \\
p! \text{ ack[release}(r)); \\
\text{empty}(r) \rightarrow \text{free}[r] := \text{true}; \\
\neg \text{empty}(r) \rightarrow \text{head.q}[r]! \text{ grant}(r); \\
\end{align*}
\]

\[
\begin{align*}
\text{process } p_j \text{ receiving an ack[release}(r)] \text{ from } p_k \\
p_k & \text{ ack[release}(r))] \\
\{ \text{no action} \}; \\
\end{align*}
\]

\[
\begin{align*}
\text{process } p_j \text{ receiving a grant for resource r from } p_k \\
p_k & \text{ grant}(r) \\
\{ \text{delete } p_k \text{ from } p_j\text{'s successor-set; } \\
\text{terminal} & := \text{true; } \\
\text{use resource } r; \\
\end{align*}
\]

\[
\begin{align*}
\text{process } p_j \text{ initiating a find_deadlock}(p_j,p_j) \text{ signal} \\
\{ \text{send } find\text._deadlock}(p_j,p_j) \text{ signal to all processes in } p_j\text{'s successor-set; } \\
\text{insert } (p_j, p_j) \text{ in } p_j\text{'s initiator-set; } \\
\text{num}_j(j) & := \text{number of elements in } p_j\text{'s successor-set; } \\
\text{pending} & := \text{true; } \\
\end{align*}
\]
{process \( p_k \) receiving a find_deadlock\((p_j,p_k)\) signal}

\[
\begin{align*}
\text{[terminal} & \rightarrow p_k! \text{ack[false}(p_j)\text{]}\text{]} \\
\neg \text{terminal} & \rightarrow \\
\text{pending} & \rightarrow \\
\quad [p_j = p_k & \rightarrow \text{BROADCAST DEADLOCK}] \\
\quad p_j \not= p_k & \rightarrow \\
\quad [(p_j, p_s) & \in p_k\text{'s initiator-set OR } p_s \in p_k\text{'s successor-set } \rightarrow \\
\quad \text{BROADCAST DEADLOCK}] \\
\quad (p_j, p_s) & \not\in p_k\text{'s initiator-set } \rightarrow \\
\quad [p_j & \in \text{any of the tuples as an initiator in } p_k\text{'s initiator-set } \rightarrow \\
\quad \text{insert } (p_j, p_s) & \text{ in } p_k\text{'s initiator-set}\] \\
\quad p_j & \not\in \text{any of the tuples as an initiator in } p_k\text{'s initiator-set } \rightarrow \\
\quad [\text{forward } \text{find_deadlock}(p_j,p_k) & \text{ to all processes in } p_k\text{'s successor-set; } \\
\quad \text{insert } (p_j, p_s) & \text{ in } p_k\text{'s initiator-set;} \\
\quad \text{num}_j(k) := \text{number of elements in } p_k\text{'s successor-set; } \] \\
\neg \text{pending} & \rightarrow \\
\quad \text{[pending} := \text{true; } \\
\quad \text{forward } \text{find_deadlock}(p_j,p_k) & \text{ to all processes in } p_k\text{'s successor-set; } \\
\quad \text{insert } (p_j, p_s) & \text{ in } p_k\text{'s initiator-set;} \\
\quad \text{num}_j(k) := \text{number of elements in } p_k\text{'s successor-set; } \]
\end{align*}
\]

\{process \( p_k \) receiving an ack[false\((p_j)\)] signal\}

\[
\begin{align*}
\quad \text{[num}_j(k) := \text{num}_j(k)-1; } \\
\quad \text{num}_j(k) = 0 & \rightarrow \\
\quad & \text{[sender! ack[false}(p_j)\text{]; } \\
\quad & \text{delete all tuples containing } p_j \text{ as the initiator from } p_k\text{'s initiator-set; } \]
\end{align*}
\]

2.2.6 Resolution

The above algorithm can be modified to handle the resolution of the deadlock cycle. For this, each process except the initiator of the find_deadlock signal, includes its own process number in the signal. Thus the process detecting the cycle knows the identities of the processes.
involved in this cycle. Once the cycle is detected, an ERASE message is propagated by the process to be aborted such that this process is deleted from the initiator-set and the successor-set of the processes involved in the cycle.

The deadlock resolution consists of the following steps:

1. The process $p_i$ detecting the deadlock is chosen as the process to be aborted. Once chosen, all signals received by $p_i$ are discarded.

2. Process $p_i$ initiates an ERASE message to all its successors, so that no process has $p_i$ in its queue or as an element in any of the tuples in its initiator-set. $p_i$ is aborted when it receives its own ERASE message back.

3. $p_i$ cancels all its requests, and releases all the resources it held.

4. Initiator-set and successor-set of $p_i$ are set to null.

One important point to note is that all the other find.deadlock signals initiated by other processes are still in the initiator-sets of the processes, thus reducing the number of future messages to detect any more cycles.

### 2.2.7 Proof of Correctness

The absence of deadlock in the signalling mechanism has already been established in sections 2.2.3 and 2.2.4. This subsection only deals with a proof of the absence of false deadlocks.

**Theorem 2.1** The proposed algorithm only detects true deadlocks and no false deadlocks are reported.

*Proof:* False deadlocks are detected when the collected global state is an inconsistent one. In the proposed algorithm, in response to a find.deadlock($p_j$, $p_i$) signal, deadlock is detected if and only if $p_j$ receives the signal back, or any process $p_k$ receives find.deadlock($p_j$, $p_i$) signal initiated by $p_j$ for the second time from process $p_i$.

Since all signals propagate along the edges of the waitfor graph, one can conclude that the initiator or one of its successor nodes may be involved in a circular waiting. If a cycle
exists and a process is abruptly terminated before broadcasting the deadlock or before passing on the \( \text{ack[false(p_j)]} \) signal to its predecessor node, then that predecessor node becomes a terminal node, and it returns an \( \text{ack[false(p_j)]} \), indicating that the deadlock does not exist. Thus only true deadlocks are detected.

End of Proof

Note: Consider the case when a process aborts itself after broadcasting the deadlock and before receiving its own ERASE message. However, in absence of a global clock, the terms before and after do not have any global significance — these are only defined by the happened-before relationship! This case therefore should not be considered as a case of a false deadlock being detected.

2.2.8 Example

Consider a distributed system with five processes (Figure 2.5). As seen in the figure, processes \( p_2 \), \( p_3 \), and \( p_4 \) are executing. \( p_1 \) is waiting for resources from \( p_2, p_4 \), and \( p_5 \) while \( p_5 \) is waiting for a resource held by \( p_3 \).

Suppose \( p_2 \) completes its execution and releases the resource to \( p_1 \). \( p_2 \) is deleted from the successor-set of \( p_1 \). Now, if \( p_3 \) needs a resource currently held by \( p_1 \) then this request would create a cycle involving processes \( p_1, p_b, p_3 \) (Figure 2.6). After a certain amount of time, \( p_1 \) initiates a \( \text{find_deadlock}(p_1,p_1) \) signal and sends it to all the processes in its successor set.

<table>
<thead>
<tr>
<th>process</th>
<th>successor-set</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( { p_2, p_4, p_5 } )</td>
<td>( \phi )</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>( { \phi } )</td>
<td>( p_1 )</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>( { \phi } )</td>
<td>( p_5 )</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>( { \phi } )</td>
<td>( p_1 )</td>
</tr>
<tr>
<td>( p_5 )</td>
<td>( { p_3 } )</td>
<td>( p_1 )</td>
</tr>
</tbody>
</table>
successor-set. On initiating the $\text{find deadlock}(p_1,p_1)$ signal, $p_1$ became a pending node and had $p_1$ in its initiator-set. Since $p_4$ is a terminal node, it sends an $\text{ackfalse}(p_1)$ back to $p_1$, while $p_5$ forwards the $\text{find deadlock}(p_1,p_5)$ signal to $p_3$, inserts $p_1$ in its initiator-set, and becomes a pending node. $p_3$ also forwards the $\text{find deadlock}(p_1,p_3)$ signal to $p_1$ and inserts $p_1$ in its initiator-set and becomes a pending node. By receiving the $\text{find deadlock}(p_1,p_3)$ signal back, $p_1$ determines the existence of a cycle and broadcasts deadlock.

The resolution of the above deadlock cycle takes place as follows: $p_1$ on detecting the deadlock cycle is chosen as the process to be aborted. $p_1$ cancels all its requests and releases all the resources it held. $p_1$ also initiates and sends an ERASE message to all its successors. Thus $p_1$ is eliminated from the queue, successor-set, and initiator-set of all the processes in the system. On receiving the ERASE message back, $p_1$ is aborted from the system. Now $p_3$ and $p_4$ will execute while $p_5$ waits for a resource from $p_3$ as shown in Figure 2.7.
2.3 Algorithm 2.3

2.3.1 Model Description

Messages sent from process A to process B are received by process B in the same order as they were sent. To detect the presence of deadlock in the proposed algorithm we do not use probe messages. Instead, we use an update message, one of its function being to check for the occurrence of deadlock.

Each site in the network carries a unique site identifier called Site.ID (Figure 2.8). Within the network a site maintains a certain portion of the database. Each site owns some data objects and maintains a few transactions. Each data object is identified by a unique identifier denoted by Data.obj. Every data object controlled by a site has a variable called Locked.by. The variable Locked.by determines the current state of the data object. If the data object is not locked by any transaction, Locked.by will store nil. Otherwise, it stores the identification of the locking transaction.

<table>
<thead>
<tr>
<th>Data Objects</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data.obj</td>
<td>Data.obj</td>
</tr>
<tr>
<td>Locked.By</td>
<td>Locked.By</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transactions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T.ID</td>
<td>T.ID</td>
</tr>
<tr>
<td>Update.message</td>
<td>Update.message</td>
</tr>
<tr>
<td>Incoming.Edge</td>
<td>Incoming.Edge</td>
</tr>
<tr>
<td>Outgoing.Edge</td>
<td>Outgoing.Edge</td>
</tr>
</tbody>
</table>

Figure 2.8: Transactions and Data Objects in Site $S_i$.

Each transaction has a unique site identifier denoted by T.ID. A transaction can use data objects within its own site or make explicit requests for a data object in another site.
As each site has a unique Site_ID, and every transaction within a site has a unique T_ID, the T_ID can be considered to be unique throughout the network.

A transaction can be in one of three states: active, blocked, or waiting. A transaction is said to be active if it is executing. It is said to be blocked when it has made a lock_request for a specific data object and it was not granted access to that object. A transaction is in a waiting state when it has made a request for a data object, but has not received the reply yet.

The data structure for each transaction \( T_i \) at site \( S_i \) are: the set \( \text{Incoming}_\text{Edge}(T_i) \), the set \( \text{Outgoing}_\text{Edge}(T_i) \), a variable called \( \text{State}(T_i) \), and a variable called \( \text{Update}_\text{message}(T_i) \). Given a transaction \( T_i \), \( \text{Incoming}_\text{Edge}(T_i) \) is the set of all transactions which have requested for a data object which is currently locked by \( T_i \). Each element in the \( \text{Incoming}_\text{Edge}(T_i) \) set is a tuple \((T_j, D_i)\) where \( T_j \) is the requesting transaction and \( D_i \) is the specific data object requested by \( T_2 \). \( \text{Outgoing}_\text{Edge}(T_i) \) is the set of all transactions to which outstanding requests are made by \( T_i \). If a transaction \( T_i \) is in active state, \( \text{Outgoing}_\text{Edge}(T_i) \) is \( \text{nil} \). \( \text{State}(T_i) \) determines the current state of \( T_i \). \( \text{Update}_\text{message}(T_i) \) will be used to store the most recent update message that \( T_i \) has received.

### 2.3.2 Explanation of the Algorithm

Suppose a transaction \( T_i \) makes a lock request for a data object \( D_j \). If \( D_j \) is free then \( D_j \) is granted to \( T_i \) and \( \text{Locked}_\text{by}(D_j) \) is set to \( T_i \). If \( D_j \) is not free then \( D_j \) sends a not_granted message to \( T_i \) along with the transaction identifier locking \( D_j \) (henceforth called \( T_j \)). \( T_i \) becomes an element in the \( \text{Incoming}_\text{Edge}(T_j) \) and \( T_j \) becomes an element in the \( \text{Outgoing}_\text{Edge}(T_i) \). Now \( T_i \) initiates an update message with \( T_j \) as its parameter to modify all the \( \text{Update}_\text{message} \) variables which are affected by the changes in \( \text{Locked}_\text{by} \) variable of the data objects. \( \text{Update} \) message is a recursive function call that will continue updating all elements of every \( \text{Incoming}_\text{Edge} \) in the chain.

When a transaction \( T_i \) receives an update message, it sets its \( \text{Update}_\text{message} \) to the new value. Now, a check for deadlock is performed. If a deadlock is not detected then the update message is forwarded. Otherwise, deadlock is declared and deadlock resolution is initiated.
The transaction detecting the deadlock is chosen as the one to be aborted. This transaction sends a clear message to the transaction(s) holding its requested data object. It also allocates every data object it held to the first requester in its Incoming.Edge and enqueues remaining requesters to the new transaction.

The transaction receiving the clear message purges the tuple in its Incoming.Edge having the aborting transaction as an element.

{Transaction $T_i$ makes a lock_request for data object $D_j$}
begin
  send lock_request($T_i$) for $D_j$;
  set State($T_i$) to waiting;
  wait for reply;
  if granted then begin
    Locked_by($D_j$) := $T_i$;
    set State($T_i$) to active;
  end {if}
else {suppose $D_j$ is being used by transaction $T_j$}
begin
  set State($T_i$) to blocked;
  add $T_j$ to Outgoing.Edge($T_i$);
  send update($T_j$) to every element of Incoming.Edge($T_i$);
end; {else}
end;

{Data object $D_j$ receiving a lock_request($T_i$)}
begin
  if Locked_by($D_j$) = nil then
    send granted to $T_i$
  else begin
    send not.granted($T_j$) to $T_i$;
    add ($T_i$, $D_j$) to Incoming.Edge($T_j$);
  end; {else}
end;

{Transaction $T_j$ receiving an update($T_i$) message}
begin
  Update.message($T_j$) := $T_i$;
  if (Incoming.Edge($T_j$) ≠ nil) then
    if ((Outgoing.Edge($T_j$) \ Incoming.Edge($T_j$) = nil) and
      (Update.message($T_j$) ∈ any of the tuples of Incoming.Edge($T_j$))) then
      send update(Update.message) to every element of Incoming.Edge($T_j$)
    else begin
      DECLARE DEADLOCK;
      {initiate deadlock resolution}
      {$T_j$ is chosen as the transaction to be aborted}
\{T_j \text{ releases all the data objects it holds}\}
send clear(T_j) to every element of Outgoing\_Edge(T_j);
allocate each data object \(D_i\) held by \(T_j\) to the first requester
\(T_k\) in Incoming\_Edge(T_j);
for every transaction \(T_j\) in Outgoing\_Edge(T_j) \text{ requesting data object } D_i, add \((T_i, D_i)\) to Incoming\_Edge(T_k);
\}
end; \{else\}
\}
end;

\{Transaction T_k \text{ receiving a clear(T_j) message}\}
begin
  purge the tuple having \(T_j\) as the requesting transaction from Incoming\_Edge(T_k);
end;

\subsection*{2.3.3 Proof of Correctness}

\textbf{Theorem 2.2} \textit{All true deadlocks are detected.}

\textit{Proof:} Assume a cycle could be created such that a deadlock would not be detected. Clearly, a simple cycle with only two transactions involved cannot satisfy the assumption, because Incom\_Edge \(\cap\) Outcom\_Edge \(\neq\) nil. So, the cycle must have more than two transactions. Suppose a transaction \(T_i\) makes a request which creates a cycle. \(T_i\) will be an element of some Incom\_Edge(T_j), \(T_i\) will begin to propagate an update message. Let Update\_message(T_i) be equal to \(T_j\). Since a cycle exists, all the Incom\_Edge sets will have at least one element. At some point, \(T_j\) will receive update message and will propagate it to all elements of its Incom\_Edge. Now, a deadlock is detected. Therefore, the initial assumption cannot be true. So, all true deadlocks are detected.

\textit{End of Proof}

\textbf{Theorem 2.3} \textit{No false deadlocks are detected.}

\textit{Proof:} As has been described earlier, if there are no cycles, the update message stops at the transaction having an empty Incom\_Edge. If there was a cycle, then either \((\text{Outcom\_Edge(T_j)} \cap \text{Incom\_Edge(T_j)} = \text{nil})\) or \((\text{Update\_message(T_j)}\) is an element of some tuple of Incom\_Edge(T_j)) would have been true, thus making \(T_j\) to declare the deadlock. Hence, the proposed algorithm prevents the declaration of false deadlocks.

\textit{End of Proof}
2.3.4 An Example

Consider a distributed database with seven transactions as shown in Figure 2.9. The table in Figure 2.9 gives the details of the sets \text{Incoming	extunderscore Edge} and \text{Outgoing	extunderscore Edge}. At this instance, there are no update messages. As shown in the figure, a process can have multiple outstanding requests for resources. As the processes join the system, they initiate update messages. But no cycle is formed. Hence these processes cannot declare any deadlock.

Suppose, \( T_6 \) makes a request for data object currently held by \( T_1 \). This request will create two cycles, one involving transactions \( T_1, T_2, T_3, T_5, T_6, \) and \( T_1 \) and the other involving transactions \( T_1, T_2, T_4, T_5, T_6, \) and \( T_1 \) as shown in Figure 2.10. Once \( D_1 \) sends a \text{not	extunderscore granted}(T_1) message back to the request from \( T_6, \) and the respective \text{Incoming	extunderscore Edge}(T_1) and \text{Outgoing	extunderscore Edge}(T_6) are updated, \( T_6 \) initiates an \text{update} message with \( T_1 \) as its parameter and sends it to the process in its \text{Incoming	extunderscore Edge} (in this case \( T_6 \)). \( T_5 \) checks for possible occurrence of deadlock and forwards the \text{update} message to \( T_3 \) and \( T_4 \). The process of forwarding continues until the message reaches \( T_2 \). \( T_2 \) while checking for occurrence of deadlock finds that \text{Update	extunderscore message}(T_2) is an element of a tuple of \text{Incoming	extunderscore Edge}(T_2) and declares deadlock. Now \( T_2 \) is chosen as the transaction to be aborted. \( T_2 \) releases all data objects it holds (if any) and sends a \text{clear} message to all transactions in its \text{Outgoing	extunderscore Edge}. Each of the transactions in \text{Outgoing	extunderscore Edge}(T_2) purges the tuple containing \( T_2 \) as an element from its corresponding \text{Incoming	extunderscore Edge}'s. Then each of the data objects held by \( T_2 \) is allocated to the first requester in the \text{Incoming	extunderscore Edge}(T_2). In this case, if \( T_2 \) held any data object, it is allocated to \( T_1 \) if that particular data object was requested by \( T_1 \). If there were more than one element in the \text{Incoming	extunderscore Edge}(T_2) requesting the same data object, then the remaining transactions are put in the \text{Incoming	extunderscore Edge} of the transaction that now received the data object.

2.4 Simulation

About the performance of the distributed deadlock detection algorithms, very little has been reported. Most of the papers discuss the performance issues analytically and the performance is discussed in terms of only the number of messages, storage, and delay involved.
Figure 2.9: A distributed database system with seven transactions.
Figure 2.10: The system after cycles are formed.

<table>
<thead>
<tr>
<th>T_ID</th>
<th>Incoming_Edge</th>
<th>Outgoing_Edge</th>
<th>Update_message</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>(T1, D0)</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>T1</td>
<td>(T6, D1)</td>
<td>T0, T2</td>
<td>nil</td>
</tr>
<tr>
<td>T2</td>
<td>(T1, D2)</td>
<td>T3, T4</td>
<td>nil</td>
</tr>
<tr>
<td>T3</td>
<td>(T2, D3)</td>
<td>T5</td>
<td>nil</td>
</tr>
<tr>
<td>T4</td>
<td>(T2, D4)</td>
<td>T5</td>
<td>nil</td>
</tr>
<tr>
<td>T5</td>
<td>(T3, D5), (T4, D5)</td>
<td>T6</td>
<td>nil</td>
</tr>
<tr>
<td>T6</td>
<td>(T5, D6)</td>
<td>T1</td>
<td>nil</td>
</tr>
</tbody>
</table>
in detecting a deadlock. The number of messages involved in detecting a deadlock depends on the number of sites involved in the deadlock cycle and also on the frequency of initiating the deadlock detection algorithm. If the computation is initiated rarely, the delay involved in detecting the cycle would increase and the throughput of the system decreases.

There are many ways in which the simulation of the distributed deadlock detection algorithms can be done. One of the ways being to implement them on a real system, which would be very expensive and time consuming. Another way is to evaluate their performance by simulating a distributed environment. We study the performance of the algorithms by implementing them under a distributed environment and testing the algorithms for various sets of values. Our method assumes no limitations on the processing capabilities.

The problem with all the three algorithms considered is that if the initiator of the deadlock message (or probe message) is outside a cycle, this particular cycle is not detected by this message. In Chandy, et. al.'s algorithm[3] and Algorithm 2.2 it is assumed that eventually one of the processes in this cycle would initiate a deadlock message and the cycle is detected, whereas in the case of Choudhary, et. al.'s algorithm[5], a probe is initiated only when an antagonistic conflict occurs. Readers are referred to the papers [3, 5] and Algorithm 2.2 (Section 2.2) for further details.

Algorithm 2.2 (Section 2.2) detects a cycle when only two processes are involved in a cycle and the initiator of the find_deadlock message is outside the cycle.

2.4.1 Method

The implementations described are running on a network of SparcStation 1's running SunOS 4.0.3c, which is a Berkeley 4.3BSD based UNIX system. They utilize primitives built upon the reliable and in-order message passing capabilities of the stream sockets that are available in 4.3BSD based UNIX. These algorithms have also been successfully executed on a Cray Y-MP, but the results are taken from the SparcStation version.

When a process begins, it makes a list of resources that it will request. This list is chosen randomly and is taken from all the possible resources. After this list is created, the process initializes the operating system specific code, and waits for messages to arrive. To make sure that all processes start requesting resources at the same time, a message is sent from a
process which initializes the algorithm to each process indicating when to begin its requests. Since the processes are created sequentially and some processes begin executing sooner than others, the 'start' message is needed to ensure that the quicker processes would not request and be granted all of their resources and release them before the other transactions had even begun, thus deadlock would never occur and the algorithms would remain untested. This 'start' message is not counted as one which is related to deadlock detection. The request, grant, wait, and release messages are not counted as deadlock related messages (this includes the forwarded requests and releases in [3] which are discussed below). Since a 'start' message might arrive at a process after a message from another process had arrived, any message causes a transaction to be 'activated'.

A transaction is a collection of requests and the computation needed by a process to finish its underlying computation. Since the purpose here is to simulate deadlock algorithms, when a transaction is granted all of its requests it immediately releases its resources. Each transaction consists of two phases: a request phase and a release phase. During the request phase the transaction requests a random number of resources, with the number of requests referred to as the transaction size. If all the resources are allocated, the transaction begins the second phase and releases the allocated resources, but keeps exchanging messages as needed for the algorithm. If a deadlock occurs, it is possible that some or all of the transactions will not reach the second phase. In the case of [3, 5], additional processes are created to act as resource controllers. Since the processes control the resources in the case of Section 2.2, the controller processes are not used.

2.4.2 Implementation of Chandy, et. al.'s algorithm

The implementation of Chandy, et. al.'s algorithm[3] creates two types of processes: controllers and transactions. The controllers control the resources in the system and are responsible for sending all but one of the deadlock related messages. The exceptional message, the idle message, is explained in the following paragraph. When a transaction requests a resource from a controller, the controller replies with a grant or a wait message. A resource is granted when no other transaction holds the resource. If the resource is already held by a transaction then a wait is sent to the requester. Each transaction is assigned to a controller
and becomes a local transaction for that controller. If a transaction requests a resource that its controller does not control then the controller forwards the request to the proper controller with the reply routed through the requesting transaction’s controller. As far as the transactions are concerned, their controller controls all the resources in the system. Not only does this simplify coding of the transaction, it gives each controller more information about the state of its local transactions.

The only message sent to the controller by the transaction that is related to deadlock detection is the idle message. A transaction sends the idle message when it has received a grant or wait for each request and the number of wait messages received is greater than zero. The transaction does not, however, send the idle message as soon as it becomes idle. It waits for an amount of time, $T$ [14], so that it can be reasonably sure that it will not receive grants for the remainder of its pending requests. The time to wait to send the idle message is set to a value that is dependent on the environment. Each time a grant message is received, the timer is reset to its original value, in case the grant is the precursor of several resources released at once by a completed transaction.

The controllers only initiate a probe computation when they have received a release or an idle message from every local transaction and at least one transaction is idle. When the controller knows the status of all the transactions, it begins the probe computation on behalf of all the local transactions that sent an idle message. An additional process is created to count the number of messages needed to detect deadlock. The information is only collected by this process when messages are no longer being sent or received, which can occur if there is a deadlock amongst all the uncompleted transactions or if all the transactions have committed.

Each controller keeps a list of transactions for each local resource that are waiting on the resource. The first transaction in the list is the transaction that currently holds the resource and has been sent a grant message. If the resource is released by the holder the next transaction is sent a grant (through its controller if it is a remote transaction). A list is kept that indicates which transactions are waiting on local transactions and which local transactions are waiting on remote transactions. The idle messages received are also kept in a list until the controller has received either an idle message or a release message from
every transaction.

The implementation employs the use of two message types not given in [3]. The messages are used to update the information held by a transaction regarding the release and subsequent grant to the next transaction waiting for the resource, referred to as the new holder. The first message informs the controller of the new holder of the transactions that are waiting on it to release the resource. The controller needs this information to correctly handle the receipt of a probe message. The second message type informs the controllers of the processes that are waiting on the new holder. This information is needed for the initiation of the probe computation.

2.4.3 Implementation of Choudhary, et. al.’s algorithm

The implementation of Choudhary, et.al’s algorithm[5] creates two types of processes: data managers and transactions. The data managers control the resources and the transactions attempt to allocate the resources by sending request messages to the data manager, as in the [3]. However, the transactions in [5] are more aware of their environment. They participate extensively in the deadlock detection algorithm and they must send resource requests directly to the data manager which controls the resource. When a transaction waits for a reply (either grant or wait) for a request, it does not make additional requests. When a wait is received it determines itself to be idle and sends a copy of its probe queue to the data manager that sent it the wait message as given in [5]. The deadlock resolution portion of the algorithm is not implemented to ensure that the results are similar to other implementations which do not have a means of resolution. As in the implementation of [3], an additional process is used to count the number of messages sent that are used for deadlock detection.

Each data manager keeps a list of transactions for the resource it manages with the first transaction in the list being the holder of the resource. A list of the victims of a deadlock resolution is also kept by the data manager so that when a probe message is received by the transaction it can be ignored. Each process keeps a list of the probe messages that it has received so it can resend them to each data manager that sends it a wait message.
2.4.4 Implementation of Algorithm 2.2

The implementation of Algorithm 2.2 utilizes only one type of process. This process must control one resource as well as act as a transaction that must allocate resources. The transaction makes multiple requests which are replied to with either a grant or a wait message. A transaction will initiate a probe (find_deadlock) message only after it becomes idle, when it receives a wait or grant for every request and at least one wait. It then waits to be reasonably certain that it will not receive grants causing it to no longer be idle. If the transaction does not receive more grants before the wait time is completed, it will send probe messages to each of the transactions on which it is waiting, starting the deadlock computation. This implementation also utilizes an additional process to collect information about how many messages are needed to detect deadlock (if any).

Each process keeps a list of transactions that are waiting on the resource controlled by the process. As before the transaction that is first in the list holds the resource.

2.4.5 Parameters

In this subsection, we briefly describe the parameters of the simulation of the three deadlock detection algorithms.

Input Parameters

No. of Sites This parameter represents the total number of sites (controllers) in the distributed database system.

No. of Resources The number of resources in the total system.

No. of Transactions This parameter denotes the total number of transactions in the system at any time.

Output Parameters

No. of Messages for Deadlock detection The parameter represents the number of messages involved in detecting a deadlock, if any in the system. This includes the local and intersite messages.
Average Deadlock Length When a deadlock is detected, there involves at least two transactions (processes). This parameter gives the average deadlock length for cycles of different lengths.

2.4.6 Results

In this section, we present the results of the simulation of the three algorithms considered. Due to the processing limitations, our test cases were executed for a maximum of 100 concurrent transactions. The tests were performed with various sets of input data, with each set of values tested several times to attain the average performance, which was plotted on a graph. Our tests were conducted mainly by varying the average transaction size. Some more tests are currently being performed.

The graphs are plotted using the packages spline and xgraph. Due to the scaling limitations, the graphs of [5] are given separately from those of [3] and Algorithm 2.2.

Figures 2.11 and 2.12 show the number of messages involved in detecting the cycles (if any) plotted against the average transaction size (The transaction size is the average number of requests per transaction). As seen from Figure 2.11, Algorithm 2.2 takes far few messages for higher transaction sizes compared to Chandy, et. al.'s algorithm[3].
No. of Messages vs. Average Transaction Size

Figure 2.11: Number of Messages vs. Average Transaction Size for Chandy, et. al. and Algorithm 2.2
Figure 2.12: Number of Messages vs. Average Transaction Size
for Choudhary, et. al.'s algorithm
2.5 Conclusion

Primitive state collection process can lead to the detection of false deadlocks, since the collected global state may be an inconsistent one. Such algorithms are therefore of little practical use in distributed deadlock detection. By propagating the probe signals along the edges of the waitfor graph, a process can derive consistent information about the terminal state. The algorithms [3, 9] belong to the latter category. The proposed algorithms are also based on the concept of chasing the edge of the waitfor graph, and has similarity with the work in [3]. The emphasis of the algorithms however lies in the foundation of a strong signalling mechanism used in a supervisory capacity.

Every signal in Algorithm 2.2 is treated as an interrupt, and the prioritization of these signals (Figure 2.4) leads to a better coordination of the various messages which might originate from multiple initiator nodes. The proposed algorithm (Algorithm 2.2) detects deadlock even when a subset of processes have entered into a deadlock. Careful thoughts were given to the assignment of priority levels to the different signals. For example, life would have been simpler if release were given the lowest level priority. However, it was given the highest priority because a last minute release might probably enable a node to send an ack[false] instead of an ack[true] thereby preventing a possible false deadlock detection. However, a different way to optimize these signals is not ruled out.

In Algorithm 2.3, probe messages are not used to detect deadlock. The function of update message is two fold: first to modify the Update.message variable, then to check the occurrence of deadlock. Also, as there is no single central site involved to maintain global information, the system described here is less prone to failure. In the worst case, the overall message complexity of the above algorithm is $O(mn)$, where $m$ is the maximum number of resource requests made by any transaction, and $n$ is the number of transactions in the system. Possible extensions to the algorithm could easily tackle the problems of shared and exclusive locks.

In comparison with other deadlock detection algorithms, the proposed algorithms have the following advantages:

1. The process detecting the deadlock cycle immediately broadcasts, thus reducing the
message complexity and detecting the most frequent deadlocks.

2. The proposed algorithm takes far few messages for higher transaction sizes compared to Chandy, et. al.'s algorithm[3].

3. Since no central site is involved to maintain the global information, the graph is less prone to failure.

The performance results for three distributed deadlock detection algorithms are also presented. The purpose of this simulation was to study the performance issues of Algorithm 2.2 compared with two other well-known algorithms. Most of the existing papers discuss the performance issues analytically in terms of the number of messages, storage, and delay involved in detecting a deadlock. One common problem that was noticed with all the three algorithms considered was that if the initiator of the probe (find_deadlock) message is outside the cycle, then this cycle goes undetected with this particular message. In a special case, Algorithm 2.2 detects such a cycle.

The implementation results show that in many situations (especially the number of messages involved in detecting the cycles), Algorithm 2.2 performs much better compared to Chandy, et. al.[3] and Choudhary, et. al.[5]'s algorithms. Due to the processing limitations, our test cases were executed for a maximum of 100 concurrent transactions.
Chapter 3

DEADLOCK PREVENTION

This chapter deals with a new method of preventing deadlocks in resource sharing for distributed systems. The algorithm[7] is based on the notion of coloring the nodes of the waitfor graph, and is built on a signalling mechanism which can be implemented on an underlying routing protocol. This algorithm supports multiple resources and multiple outstanding requests. Proof of correctness of the algorithm is also presented.

3.1 Related Work

Minoura[31] gives complete details about the work done in deadlock prevention in centralized shared memory systems. In the distributed environment a lot of papers have been published in the area of deadlock detection than prevention. Readers interested in deadlock detection papers are referred to surveys in [10, 26, 35]. The problem that arises in the case of deadlock prevention algorithms with a centralized memory systems is that if the central site fails then the entire system breaks down. As discussed in [10, 26, 35], in many of the deadlock detection and prevention algorithms, some deadlocks are never detected which defeats the purpose of such an algorithm, while some algorithms detect false deadlocks thus increasing the number of rollbacks, and some other algorithms adopt an overcautious approach in handling the resource requests leading to unnecessary rollbacks. Some of these algorithms use resource ordering [16], process numbering, priorities of processes [33]. In
[25], a scheme similar to the dynamic priority based scheme is used. The rollback decision is made on the ranking of the nodes of the waitfor graph. The reranking of the node is done dynamically. But this scheme may still cause unnecessary rollbacks due to the inconclusive decision about possible reranking of the nodes.

3.2 Basic Concepts

The processes on these sites can use resources or share information local to them or available over the network. The processes may request for resources in any order which is not known a priori. The requested resources may be available or locked by other processes, thus building a graph called wait-for graph. More precise explanation of such graphs was given in Section 2.2.2. Prevention of deadlock amounts to the prevention of the occurrences of cycles in the waitfor graph.

In our algorithm each process $p_k$ maintains three variables: one queue and two sets. These are defined below:

**Definition 5** A resource-queue of a process $p_k$ contains the processes waiting for and using the resource owned by $p_k$. This queue is maintained in first-in-first-out order.

The function $\text{head}(\text{resource-queue})$ returns the process using the resource and the function $\text{tail}(\text{resource-queue})$ returns the process which made the request most recently (henceforth referred as last process). Note that one process owns maximum one resource. This assumption is made to simplify the algorithm. The WFG would be very complicated for the case of multiple resources owned by a process. At this point, we introduce the concept of the color of a node. Every node in the waitfor graph has a color which is an element of the set of process numbers $\{1, 2, 3, \ldots, n\}$ in the universe of processes, and is defined as follows:

**Definition 6** A wait-set of a process $p_k$ is a set of tuples of the form $(p_i, p_j)$ where $p_i$ is the process that has requested for a resource from $p_j$, and $p_k$ is the last process in the queue waiting for the resource held by $p_j$.

The wait-set is used to build the WFG.
Figure 3.1: The building of a waitfor graph. (a) $p_4$ requests for a resource owned by $p_3$. (b) This resource is currently being used by $p_7$ and $p_5$ is the last process in the queue for this resource. (c) $p_5$ sends an ack to $p_4$.

**Definition 7** A dependent-set of a process contains all the owners (processes) of the resources it has requested. Once a resource is granted, the owner of that resource is deleted from the dependent-set.

### 3.2.1 Building a signalling mechanism

Building of the signalling mechanism differs as the data structures are quite different from the deadlock detection algorithms.

When a process $p_i$ needs a resource owned by another process $p_j$, it sends a request to $p_j$ and sets up a provisional arc from $p_i$ to $p_j$. If the resource is available, then the request is granted to $p_i$ and the provisional arc is removed. If on the other hand, the resource is not available, then a message is forwarded to the last process $p_k$ which requested for that resource, and the provisional arc from $p_i$ to $p_j$ is converted to a provisional arc from $p_i$ to
The process $p_k$ eventually decides whether it would allow $p_i$ to wait for that resource. A positive decision is communicated to $p_i$ by sending an *ack* (acknowledgment) signal via $p_j$, and the provisional arc from $p_i$ to $p_k$ is converted to a true arc from $p_i$ to $p_k$. When $p_k$ sends an *ack*/*request* to $p_i$, a tuple $(p_i, p_j)$ is included in $p_k$'s *wait-set* and $p_j$ is added to the *dependent-set* of $p_i$. If the decision is negative, then a *nack* (no acknowledgment) signal is sent to $p_i$ via $p_j$. The provisional arc from $p_i$ to $p_k$ is removed, and $p_i$ rolls back.

Figure 3.1 shows building of a WFG using the signalling mechanism. The request made by $p_4$ to $p_3$ creates an arc $(p_4, p_3)$ in the WFG. One new tuple $(p_4, p_3)$ is included in $p_3$'s *wait-set* and $p_3$ is added to the dependent-set of $p_4$.

![Figure 3.2](image_url)

Figure 3.2: Example of a process rollback. (a) process $p_1$ makes a request for a resource to process $p_3$. (b) process $p_1$ rolls back.

In Figure 3.2, $p_1$ makes a request to $p_3$. $p_3$ sends a *nack* to $p_1$, thus causing $p_1$ to rollback. So $p_1$ releases all its resources and cancels all its requests. After a certain amount of time, $p_1$ can restart.

The communication of the *request* and the *ack* or *nack* signals between a pair of processes is thus channeled through the owner of the concerned resource. The waiting processes maintain the *dependent-set* containing the processes from which it has requested for the resources. When a process completes using a resource, it sends a *release* signal to the owner of that resource, which acknowledges it and reallocates that resource. This results in the removal of an arc in the WFG. A release signal is also sent when a process is preempted due to any reason. One of the ways a process can be preempted is by rollback.
Lemma 3.1 In any deadlock prevention algorithm if the maximum claims of the processes are not known in advance, process rollback is unavoidable.

Proof: No specific strategy about requesting for resources can be formulated if the claims of the processes are not known a priori. In this environment, a process $p_i$ can wait for another process $p_j$ if the resource needed by $p_i$ is currently being used by $p_j$. However, process $p_j$ may also need some resource which is currently being used by process $p_i$. To prevent a deadlock, obviously one of these two processes has to rollback.

End of Proof

The pair of signals $request$, $ack[request]$ or $nack[request]$ constitutes an atomic action in as much as every process receiving a $request$ defers the decision about a subsequent $request$ until it has sent the corresponding $ack$ or $nack$ signal.

At this point, we introduce the concept of the color of a node. Every node in the waitfor graph has a color which is an element of the set of process numbers $\{1, 2, 3, \ldots, n\}$ and is defined as follows:

Definition 8 In a WFG, the color of a terminal node is the process number corresponding to that node. For all other nodes, the color is the same as the color of its successor node in the graph. If a node has more than one successor, then the highest color of the successors is the color of this node.

Figure 3.3 shows the color of each node in the sample waitfor graph.

Note 1: A node could also choose the lowest color of its successors as its color.

The above Definition 8 makes an additional assumption that each node in the WFG knows its color. How does a node know about its current color? This calls for two additional signals in the signal repertoire of each process: $get\ color$ and $ack[\ color]$. Colors are not static - everytime an arc is created or removed in the waitfor graph, the color of many processes could change. It is important to observe that when a process has to make a decision about sending an $ack$ or a $nack$ in response to a $request$, it should not depend on the stale information about its own color, but try to collect the latest information. Therefore, prior to such a decision, a process issues a $get\ color$ signal to its successor nodes. This signal eventually propagates upto the present terminal nodes of the WFG to which the process
Figure 3.3: A waitfor graph with each node labeled by its present color.

belongs. An active process receiving a request for a resource propagates a get color signal in its WFG. If the node has not yet received a reply (ack[request] or nack[request]) for its request, it marks its own number as the color of the requesting node, and sends it with the help of an ack[color] signal back to the node which issued the get color signal. To avoid complications, consider the pair of signals (get color, ack[color]) to constitute another atomic operation.

A process can be in one of three states: active, inactive and executing. A process said to be active if a process in the WFG has made a request for a resource and has not received any response yet. A process is said to be inactive if no process in the WFG has made a request. A process is in the executing state if it has no outstanding request, i.e., has no outgoing edges in the WFG.

Note 2: Only the terminal nodes can be in the executing state.

Note 3: A terminal node is in both active and executing state if it has no outstanding request and some other node in the WFG has made a request.

Note 4: A terminal node is in both inactive and executing state if it has no outstanding request and no node in the WFG has made a request.

Finally, before sending a request, the node sends an activate signal to all nodes in its
WFG. Only after the corresponding \textit{ack[activate]} signal is received from all the nodes in that WFG, the \textit{request} is made. After a node receives an \textit{activate} signal, it becomes \textit{active} and remains in that state, until it receives a \textit{deactivate} signal. The \textit{deactivate} signal is sent by the requesting node to all nodes in the WFG when it receives either an \textit{ack[request]} or a \textit{nack[request]} in response to its own \textit{request}. Each node confirms the deactivation by returning an \textit{ack[deactivate]} signal up the WFG back to the requesting node, and switches to the \textit{inactive} state. The activation and the deactivation can be conveniently performed using the algorithm due to Dijkstra and Scholten[9], where signals and acknowledgments propagate along the edges of the waitfor graph.

The set of signals introduced so far can be summarized as follows:

1. request, \textit{ack[request]} or \textit{nack[request]}
2. get color, \textit{ack[color]}
3. activate, \textit{ack[activate]}, deactivate, \textit{ack[deactivate]}
4. release, \textit{ack[release]}

These signals are supervisory signals and are distinct from the usual interprocess communication signals, which are an integral part of the underlying computation.

### 3.3 The Algorithm

The crucial part of the algorithm is the generation of an \textit{ack} or a \textit{nack} signal in response to a \textit{request} for a resource. The foundations are laid by the following three lemmas.

**Lemma 3.2** For any node in a WFG receiving a request, if the color of the requesting node is the same as the color of its own (the node receiving the request), then sending an \textit{ack} signal may create a cycle.

\textit{Proof:} If the requesting and the requested nodes have the same \textit{color}, these two nodes belong to the same WFG. If an \textit{ack} is sent to the requesting node, one new edge will be formed in the same WFG. This new edge will create a cycle if there exists a directed path from the requested node to the requesting node in the original WFG (before the new edge is formed).

\textit{End of Proof}
Remark 1: If the color of the requesting and the requested node are the same, sending an ack will not create a cycle if there is no directed path from the requested node to the requesting node in the WFG, all paths containing these nodes were directed from the requesting node to the requested node.

From Lemma 3.2 and Remark 1, it is obvious that if the color of the requesting node is the same as the color of its own, a nack signal must be sent to prevent the occurrence of deadlock.

Lemma 3.3 For any inactive node in a WFG receiving a request, if the color of the requesting node differs from the color of its own, then sending an ack signal never forms a cycle.

Proof: If there is an inactive node in a WFG, then no node in the WFG has made a request. If this inactive node receives a request, obviously, the requesting node belongs to a different WFG. If an ack is sent to the requesting node, it will create a new edge and this edge will join the two WFGs. But, in no case, this new edge can form a cycle, since minimum two edges are necessary to form a cycle between two disjoint WFGs.

End of Proof

Lemma 3.4 For any active node in the WFG receiving a request,

(i) if the color of the requesting node is greater than the color of its own, then sending an ack signal, and

(ii) if the color of the requesting node is less than its own color, then sending a nack signal never creates a cycle.

Proof: An active node $p_i$ in a WFG knows that some node in the WFG has sent a request. However, $p_i$ does not know to whom the request has been sent. If this request has been sent to some node belonging to the same WFG to which the requesting node belongs, then an unconditional ack might create a cycle. So, a sequential ordering among the nodes according to their colors will avoid the cycle. The conditions (i) and (ii) allow an arc to be formed from a node of higher color to a node of lower color and does not allow an arc to be formed in the opposite direction. This prevents the creation of a cycle.
Based on the above lemmas, the detailed algorithm for deadlock prevention is given below:

**Procedure send_request;**
{process $pj$ sending a request to $pk$; process $pj$ wants a resource owned by $pk$}

**begin**
  send *activate* to every node in the WFG;
  wait for *ack*[activate] from every node;
  send request for a resource to $pk$ and wait for *grant*, *ack* or *nack*;
  if *grant* is received then send *deactivate* signal to every node;
  if *ack* is received then
    **begin**
    send *deactivate* to every node;
    insert $pk$ in $pj$’s dependent-set;
    send *get color*; {change color since the successor is changed} wait for *ack*[color];
    wait for the resource;
  **end**
  else if *nack* is received then
    **begin**
    send *deactivate* to every node;
    rollback;
  **end**;
**end;** {send_request}

**Procedure receive_request;**
{process $pi$ receives a request for a resource from $pj$; $pk$ is the owner of the resource}

**begin**
  if $pi = pk$ then {$pi$ is the owner of the resource}
    if resource is free then
      send *grant* signal
    else if resource-queue not empty then
      **begin**
      forward request to tail(resource-queue);
      exit;
      **end**;
    send *get color*;
  wait for *ack*[color];
  own color = max(color of processes in dependent-set);
  if own color = request color then send *nack*;
  else
    if inactive then send *ack*;
  else if own color < request color then
    **begin**
    enqueue $pj$ in $pk$’s resource-queue;
Procedure receive_request;
{process pj receives a request signal for a resource from pk}
begin
    if pj in pk's wait-set then
        insert (pj,pk) in pj's wait-set; {add an edge (pj, pk) in the WFG}
        send ack;
    else
        send nack;
end; {receive_request}

Procedure receive_grant;
{process pj receives a grant signal for a resource from pk}
begin
    delete pk from pj's dependent-set;
    send get color; {change color since the successor is changed}
    wait for ack[color];
    if dependent-set empty then execute
    else wait for other resources;
end; {receive_grant}

Procedure send_release;
{process pj releases a resource owned by pk}
begin
    send release to pk;
    wait for ack[release];
    if ∃ pi | (pi,pk) ∈ pj's wait-set then remove (pi,pk) from wait-set;
    {remove the the edge (pi,pj) from the WFG}
end; {send_release}

Procedure receive_release;
{process pk receives a release signal from pj; pk is the owner}
begin
    dequeue pj from resource-queue;
    send ack[release] to pj;
    if resource-queue not empty then send grant to head(resource-queue);
end; {receive_release}

Procedure rollback;
{process pj has to rollback}
begin
    for every process pk in pj's dependent-set
        call send_release; {remove all requests}
    for every owner pk of the resources held by pj
        call send_release; {release all resources}
end; {rollback}

Note 5: A node sends a request to the owner (process) of the desired resource and then this request message is forwarded to the last process in the resource_queue of the owner (see Section 3.2.1). So, a node receiving a request signal may be in two different situations, the
owner or the last process in the resource_queue. These two situations are reflected in the procedure receive_request.

**Theorem 3.1** The algorithm prevents the occurrence of deadlock.

**Proof:** The proof follows from lemmas 3.2, 3.3, and 3.4.

### 3.3.1 Example

Consider a distributed system consisting of ten processes. As seen in Figure 3.4, processes $p_3$, $p_4$, $p_5$, $p_7$, and $p_9$ are executing at this time. $p_1$ and $p_2$ are waiting for a resource from $p_3$. $p_6$ and $p_{10}$ are waiting for resources from $p_5$, $p_7$, and $p_8$, while $p_8$ is waiting for a resource from $p_9$. The colors of the processes are also shown in the figure. The dependent_set, resource_queue and wait_set of the processes in the current state (Figure 3.4) are shown in Figure 3.5.

Suppose, $p_6$ makes a request for a resource to $p_9$ (Figure 3.6). Since the colors of the two processes are the same (9 in this case), a nack is sent to $p_6$ (as per Lemma 3.2). This is the case of an unavoidable rollback.

$p_9$ makes a request for a resource to $p_6$ (Figure 3.7). A cycle may occur involving processes $p_6$, $p_8$, and $p_9$ (Figure 3.7), but as the colors of $p_6$ and $p_9$ are the same (9 in this case), a nack is sent to $p_9$, making $p_9$ to rollback (as per Lemma 3.2).
<table>
<thead>
<tr>
<th>process</th>
<th>dependent-set</th>
<th>resource-queue</th>
<th>wait-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>{ p3 }</td>
<td>φ</td>
<td>{ φ }</td>
</tr>
<tr>
<td>p2</td>
<td>{ p3 }</td>
<td>φ</td>
<td>{ (p1, p3) }</td>
</tr>
<tr>
<td>p3</td>
<td>{ φ }</td>
<td>p2, p1</td>
<td>{ (p2, p3) }</td>
</tr>
<tr>
<td>p4</td>
<td>{ φ }</td>
<td>φ</td>
<td>{ φ }</td>
</tr>
<tr>
<td>p5</td>
<td>{ φ }</td>
<td>p6, p10</td>
<td>{ (p6, p5) }</td>
</tr>
<tr>
<td>p6</td>
<td>{ p5, p7, p8 }</td>
<td>p4</td>
<td>{ (p10, p5), (p10, p7), (p10, p8) }</td>
</tr>
<tr>
<td>p7</td>
<td>{ φ }</td>
<td>p6, p10</td>
<td>{ (p6, p7) }</td>
</tr>
<tr>
<td>p8</td>
<td>{ φ }</td>
<td>p6, p10</td>
<td>{ (p6, p9) }</td>
</tr>
<tr>
<td>p9</td>
<td>{ φ }</td>
<td>p8</td>
<td>{ (p8, p9) }</td>
</tr>
<tr>
<td>p10</td>
<td>{ p5, p7, p8 }</td>
<td>φ</td>
<td>{ φ }</td>
</tr>
</tbody>
</table>

Figure 3.5: The status of the variables maintained at each node.

Figure 3.6: Example of an unnecessary rollback. p6 sends a request to p9, p9 sends a nack and p6 rolls back.
Figure 3.7: Creation of a cycle is avoided. $p_3$ sends a request to $p_6$. $p_6$ sends a nack to $p_9$ and $p_9$ rolls back (unavoidable rollback).

Figure 3.8: Inactive node processing a request. $p_3$ sends a request to $p_6$. $p_6$ is inactive and sends an ack.
Figure 3.9: Active node processing a request for a resource. $p_{10}$ sends a request to $p_4$. $p_4$ sends an ack. $p_3$ sends a request to $p_6$. $p_6$ is active since $p_{10}$ has not sent the deactivate signal yet. $p_6$ sends a nack to $p_3$ and $p_3$ rolls back (unnecessary rollback).

The above two cases can also be handled by Lemma 3.4. $p_3$ makes a request to $p_6$. $p_6$ is inactive as no other node in the tree (to which $p_6$ belongs) has already sent a request (Figure 3.8). As per Lemma 3.3, $p_6$ sends an ack signal to $p_3$, thus combining the two WFG's into a single WFG.

$p_3$ and $p_{10}$ simultaneously makes a request for resources from $p_6$ and $p_4$, respectively (Figure 3.9). $p_4$ processes the request and sends an ack signal to $p_{10}$, while $p_6$, active due to the request made by $p_{10}$ sends a nack signal to $p_3$ (as per Lemma 3.4). $p_3$ releases all the resources it held and cancels all its requests for resources and rolls back (again a case of an unnecessary rollback). The situation after $p_3$ rolls back is shown in Figure 3.10.

### 3.4 Starvation

The algorithm is not free from starvation. Consider a system as shown in Figure 3.11. According to Lemma 3.4, $p_3$ sends an ack to $p_6$. $p_6$ changes its color due to its new successor ($p_3$). $p_3$ receives a grant signal from $p_6$ and $p_6$ changes its color again. Before $p_6$ sends a nack signal to $p_3$, $p_6$ makes a new request to $p_3$. So, obviously $p_3$ starves.

One approach to solve the problem is to associate a timestamp variable, timestamp with each node. timestamp is implemented in such a manner that the following condition is
always true:

\[ C1: \text{timestamp}(i) > \text{timestamp}(\text{successor}(i)). \]

This condition is enforced by including the timestamp variable's value in the get color and ack[color] signals. When the ack[color] signal starts propagating from the terminal node to its predecessors, timestamp of the predecessors is set to \(1+\text{timestamp(\text{successor})}\).

When an active node \(i\) receives a request from a node \(j\) and their colors are different, \(i\) sends an ack only if \(\text{timestamp}(j) < \text{timestamp}(i)\). When \(j\) receives the ack signal, \(j\) sets its timestamp to \(1+\text{timestamp}(i)\). Then \(j\) sends its new value of timestamp to its predecessors by using the get color and ack[color] signals to satisfy the condition C1.

The only change in the algorithm is made in the procedure receive_request. The modified version of the procedure receive_request is given below:

Procedure receive_request;
\{process \(p_i\) receives a request for a resource from \(p_j\); \(p_k\) is the owner of the resource\}
begin
  if \(p_i = p_k\) then \{\(p_i\) is the owner of the resource\}
    if resource is free then send grant signal
  else if resource-queue not empty then
    begin
      forward request to tail(resource-queue);
      exit;
    end;
  send get color;
wait for \textit{ack}[\textit{color}];

own \textit{color} = \text{max}(\text{color of processes in dependent-set});

\textbf{if} own \textit{color} = request \textit{color} \textbf{then} send \textit{nack}

\textbf{else if} inactive \textbf{then} send \textit{ack}

\textbf{else if} request \textit{timestamp} < own \textit{timestamp} \textbf{then}

\hspace{1cm} \textbf{begin}

\hspace{2cm} enqueue \textit{pj} in \textit{pk}'s \textit{resource-queue};

\hspace{2cm} insert \textit{(pj, pk)} in \textit{pj}'s \textit{wait-set}; \{add an edge \textit{(pj, pi)} in the WFG\}

\hspace{2cm} send \textit{ack};

\hspace{1cm} \textbf{end}

\textbf{else} send \textit{nack};

\textbf{end}; \{\textit{receive \_request}\}

Lemma 3.4 should be changed as follows:

\textbf{Lemma 3.5} For any active node in the WFG receiving a request,

\textit{(i)} if the timestamp of the requesting node is less than its own timestamp, then sending an \textit{ack} signal,

\textit{(ii)} if the timestamp of the requesting node is greater than its own timestamp, then sending a \textit{nack} signal, and

\image
(iii) if the timestamp of the requesting node is identical to its own timestamp, then using process numbers as tiebreakers never creates a cycle.

Proof: Similar to the original Lemma 3.4.

Lemma 3.6 For any active node in the WFG receiving a request, if the timestamp of the requesting node is less than its own timestamp, then sending an ack signal prevents starvation.

Proof: Let two processes $i$ and $j$ send request to each other, and $\text{timestamp}(i) < \text{timestamp}(j)$. So, $j$ sends an $\text{ack}$ to $i$ and then $j$ changes its $\text{timestamp}$ to $1 + \text{timestamp}(i)$. Now, $\text{timestamp}(j) < \text{timestamp}(i)$. Note that this relationship between the two timestamps only changes when any one of them sends another request to the other process. Even if $i$ sends another request to $j$, $j$ will not send an $\text{ack}$ since $\text{timestamp}(i) > \text{timestamp}(j)$. So, it is process $j$'s turn to receive an $\text{ack}$ from $i$. Hence the starvation is prevented.

End of Proof

3.5 Performance

3.5.1 Process Rollback

A primary performance criteria of deadlock prevention algorithms is to minimize process rollbacks to the best extent possible. The present algorithm is not free from unnecessary rollbacks, but process rollback has been advocated with great restraint. There are two situations where unnecessary rollback may occur. One situation arises in a special case of Lemma 3.1 and is stated in Remark 1. This situation is also shown in Figure 3.6. The second situation of unnecessary rollback occurs in a special case of Lemma 3.4 and is shown in Figure 3.9. The amount of unnecessary rollbacks can be reduced if the complete chain of predecessors and successors are maintained at each node. This will increase the required space and the amount of work to update the variables at each node. However, the amount of unnecessary rollback is significantly less compared to existing algorithms like [25, 33].
3.5.2 Complexity

This algorithm supports multiple resource requests. Assume that there are at most \( n \) processes. In the worst case, each of the operations activate, get color, deactivate and the sending of their corresponding acknowledgments would involve sending \( n \) messages. The operations request and release have constant overheads. Therefore, the overall message complexity of this algorithm is \( O(n) \). If a process makes a sequence of calls as activate, ack[activate], request, ack[request], grant, release, ack[release], the constant factor of the message complexity \((O(n))\) can be high (4 to 6). But, Bernstein, et.al[1] have found that for most applications, over 90% of the WFG cycles can be expected to be of length 2. So, the message complexity in our algorithm will not be high in most of the situations.

3.5.3 Recovery

Under normal circumstances, only the terminal node(s) of a tree is eligible to release resources. An unpleasant situation might occur if a nonterminal (i.e., waiting) node is preempted due to some reason. In such a case, a node waiting for an acknowledgment signal suddenly becomes a terminal node and is unable to receive the acknowledgments. Two typical situations are illustrated in Figure 3.12. Here the node \( p_1 \) after receiving a request, has issued a get color signal which propagated upto the terminal node \( p_4 \). But, meanwhile the node \( p_7 \) has been preempted. How would the ack[color] signal come back to \( p_1 \)? The recovery action is as follows:

- If a node waiting for ack[color] becomes a terminal node, then it returns its own process number as the ack[color] signal. If a non-initiator node waiting for ack[color] becomes an initial node, then it simply absorbs the ack[color] signal and ceases to take any further action.

Similar situations may occur with the activate and deactivate signals also, and are summarized below:

- If a node waiting for ack[activate] becomes a terminal node, then it immediately deactivates itself and sends a deactivate signal to its predecessor node. If a node
waiting for \textit{ack[activate]} becomes an initial node, then it activates itself and returns an \textit{ack[activate]} signal to its successor node.

• If a node waiting for \textit{ack[deactivate]} state becomes a terminal node, then it simply absorbs the \textit{ack[deactivate]} signal. If a node waiting for \textit{ack[deactivate]} state becomes an initial node, then it deactivates itself and sends an \textit{ack[deactivate]} signal to its successor node.

With these modifications, the proposed algorithm can cope with the dynamic nature of the processes. The recovery action has not been included in the algorithm described in Section 3.3.

3.6 Conclusion

The main contribution of this chapter is to show that the overcautious approaches in some contemporary deadlock prevention techniques can be overcome at the expense of a linear growth in the message complexity. Most of the methods [16, 25, 33] use overcautious approaches in handling the resource requests leading to unnecessary rollback. The concept of \textit{color} plays a key role in avoiding the unnecessary rollback. The overhead of messages to build the signalling mechanism may not be substantial in more than 90% of the applications since the length of the cycle will be only 2 [1]. As explained in [31], lot of research work
is carried out in the field of deadlock prevention with a centralized shared memory system. The main problem with a centralized approach is vulnerability to failure at the single central site. Menasce[29] and Obermarck[32] have come up with algorithms in decentralized systems for fault recovery. In the above algorithm, we do not deal with fault recovery. In contrast to the definite ranking of the nodes [25], the above proposed algorithm handles nodes in any order. As explained in Knapp[26] and Elmagarmid[10] the problem of stale information leading to false deadlocks in Menasce[29] and Obermarck[32], is overcome in the proposed algorithm by using colors. Also, the coloring of the nodes helps in avoiding unnecessary rollbacks to a certain extent.
Chapter 4

CONCLUSIONS

This thesis deals with a very important problem in distributed systems - deadlock. Two important aspects of deadlock are focussed in this research - detection and prevention. In Chapter 2 we presented two algorithms for deadlock detection and proved them to be correct. Also, in Chapter 2, results of simulating Algorithm 2.2 with two other existing algorithms are also presented. Results clearly indicate that in many situations (especially the number of messages required in detecting cycles), our algorithm performs much better. In Chapter 3 we presented a new method of deadlock preventing techniques by using colors.

The emphasis of our detection algorithms however lies in the foundation of a strong signalling mechanism used in a supervisory capacity. Every signal is treated as an interrupt, and the prioritization of these signals (Figure 2.4) leads to a better coordination of the various messages which might originate from multiple initiator nodes. The proposed algorithm detects deadlock even when a subset of processes have entered into a deadlock.

The implementation results show that in many situations (especially the number of messages involved in detecting the cycles), Algorithm 2.2 performs much better compared to Chandy, et. al.\[3\] and Choudhary, et. al.\[5\]'s algorithms. Due to the processing limitations, our test cases were executed for a maximum of 100 concurrent transactions.

In comparison with other deadlock detection algorithms, the proposed algorithms have the following advantages:
1. The process detecting the deadlock cycle immediately broadcasts, thus reducing the message complexity.

2. The proposed algorithm can detect the most frequent deadlocks with minimum message passing.

3. Algorithm 2.2 takes far few messages for higher transaction sizes compared to Chandy, et. al.'s algorithm[3].

4. Since no central site is involved to maintain global information, the graph is less prone to failure.

Possible extensions to the algorithms could easily tackle the problems of shared and exclusive locks. Also, further improvements to decrease the number of sites which detect the same deadlock needs to be done. If only one site detects the deadlock, the overhead of recovering will be reduced, as the synchronization of deadlock recovery is not required. More work needs to be done in the area of formal verification techniques of deadlock detection algorithms.

In the proposed prevention algorithm, we do not deal with fault recovery. In contrast to the definite ranking of the nodes [25], the proposed algorithm handles nodes in any order. As explained in Knapp[26] and Elmagarmid[10] the problem of stale information leading to false deadlocks in Menasce[29] and Obermarck[32], is overcome in the proposed algorithm by using colors. Also, the coloring of the nodes helps in avoiding unnecessary rollbacks to a certain extent.
Bibliography


[34] Shaw, A. C. *The logical design of operating systems*. Prentice Hall, 1974.


[38] Tsai, W.-C. *Distributed deadlock detection in distributed database systems*. PhD thesis, University of Illinois at Urbana-Champaign, January 1982.