A Distributed Algorithm for Deadlock Detection and Resolution

Don P. Mitchell †

Michael J. Merritt †

AT&T Bell Laboratories

ABSTRACT

This paper presents two distributed algorithms for detecting and resolving deadlocks. By insuring that only one of the deadlock processes will detect it, the problem of resolving the deadlock is simplified. That process could simply abort itself. In one version of the algorithm, an arbitrary process detects deadlock; and in a second version, the process with the lowest priority detects deadlock.

1. Introduction

A system of processes is deadlocked when a cycle forms in its wait-for graph. One method of dealing with this problem is to allow deadlocks to form but to detect them quickly and abort a process to break the cycle. If the system is local to a single site, a central resource-management process can accomplish this. All other processes request resources from the manager which maintains a representation of the wait-for graph and watches for cycle formation.

There are two problems with this approach. First, the whole system is vulnerable to the failure of the management process; and second, message passing to and from the manager is expensive. If the system is distributed over many sites, these two problems are much more severe. Although message passing is becoming cheaper, the issue of fault tolerance will certainly remain.

A number of distributed algorithms for deadlock detection have been published, and they seem to fall into two categories. Those in the first category pass information about process demands in an attempt to maintain relevant parts of the global wait-for graph on each site [MENASCE79, GLIGOR80, OBERMARCK82].

The second category of algorithms was inspired by work on parallel graph algorithms [DIJKSTRA80, CHANG82]. In this category simpler messages are passed from process to process[CHANDY82, BRACHA83]. The global wait-for graph is not explicitly built up; however, a cycle in the graph will ultimately cause messages to return to their initiators thus alerting them to the existence of deadlock.

The algorithms presented in this paper fall into this second category. An important advantage of these algorithms over earlier work is that only one process in a cycle will detect the deadlock, simplifying the problem of resolving the deadlock. A second advantage rests in their simplicity. Indeed, one author implemented the first algorithm in a local database system in under an hour. They are just as simple to analyse. Even in the face of lost messages and process failures, the correctness proofs are trivial.

2. The System Model

The system can be described by the *wait-for graph*, a directed graph in which each node represents a process, and an edge indicates that one process is waiting on a resource held *exclusively* by another. Assuming each process waits on one resource at a time, the maximum outdegree of the wait-for graph will be one. The direction of the edges are from the waiting process to the process holding the desired resource.

Each node is given two labels. The first (indicated by an index in the lower half of the node) is a private label that is unique to the node thought not necessarily constant. The second label (indicated by an

† Address: AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.

index in the upper half of the node) is public. It represents a number that can be read by other processes, and the same value may appear in other nodes.

The edges and labels define the state of the system at any moment.

3. Simple Deadlock Detection

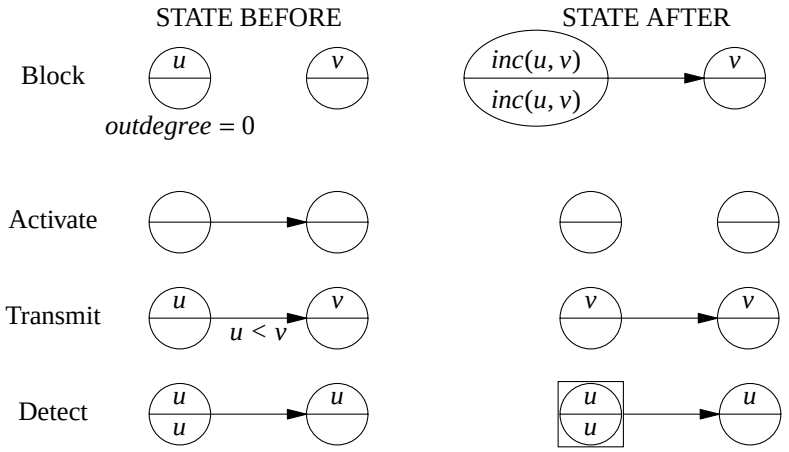


Figure 1

Figure 1 shows the four types of nondeterministic state transitions that define this algorithm. The function "*inc*(*x*, *y*)", means a value larger than both *x* and *y* that is unique to that node. Label values which neither are a precondition for a transition nor change as a result of a transition have been left blank. Each process begins with its private label equal to its public label.

The private label of each node is always unique to that node, and non-decreasing over time. These two properties can be easily realized by keeping the low-order bits of the label constant and unique while increasing the high-order bits when desired.

The *Block* step occurs when a process begins to wait on some resource held by another, creating an edge in the wait-for graph. One of the crucial features of this algorithm is the label change that occurs then. Both the public and private labels of the waiting process increase to a value greater than their previous values and greater than the public label of the process being waited on. The private and public labels of the node are changed to the same new value.

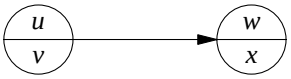
The *Activate* step means that an edge disappeared because a process got a resource, or timed out and gave up waiting, or failed. This step also occurs if the owner of a resource changes. When the waiting process notices that, it must Activate and then Block again if it is to continue waiting for that resource.

The *Transmit* step occurs when the waiting process reads the public label of the process it is waiting on and discovers that it is larger than its own. In that case, the waiting process replaces its own public label with the one it just read. One effect of this is that larger labels tend to migrate (in the opposite direction) along the edges of the wait-for graph.

The *Detect* step means that a process sees its own public label come back and knows that it is part of a cycle. A cycle of *N* processes will be detected after $N - 1$ Transmit steps. *Only one process in a cycle will detect deadlock* which simplifies the problem of resolution. The process could simply abort (or at least, release its resources) to break the deadlock, or it could initiate some other deadlock resolution scheme.

4. A Proof of Correctness

Lemma 1:



As in the figure above, if there is an edge between two nodes and $u > w$ then $u = v$.

Proof:
The definition of the Block step guarantees that this will be true when an edge first forms. The only way the label u will change during the lifetime of that edge is if a Transmit step is executed, and that will not happen as long as $u > w$, and w is nondecreasing over time. □

Lemma 2:
At the instant a cycle forms, the public labels of the nodes in it do not all have the same value.

Proof:
When the cycle forms, the last edge (like all edges) is created by a node executing the Block step. That node will have a new public label different from any other public label because the *inc* function generates unique values. □

Lemma 3:
The maximum public label value in a cycle is equal to the private label of one and only one node in the cycle.

Proof:
By Lemma 2 when a cycle forms, all the public labels cannot have the maximum value. At least one node with maximum public label value must precede a node with a different, lower public label. Thus by Lemma 1, the private label of the preceding node must equal the maximum label value. Since no Block operations can be performed by nodes in a cycle, this will remain true throughout its lifetime. Private labels are unique, so only one node can obey this condition. □

Theorem 1:
If a cycle of N nodes forms and persists long enough, exactly one node in it will execute the Detect step of the algorithm. This will happen after $N - 1$ consecutive Transmit steps.

Proof:
If a cycle forms and does not break on its own, $N - 1$ Transmit steps will carry the largest public label value all the way around the cycle. By Lemma 3, this means one and only one node will eventually execute the Detect step. □

5. Aborting Low-Priority Transactions

The algorithm just presented suffers from a significant drawback if the detecting process aborts to break deadlock. Since older processes tend to have larger label numbers, they are more likely to become the detecting process in a cycle. A designer may wish to assign a unique *priority,* p_i, to each process i, on the basis of age, number of locks held, or other criteria, and select the process with the lowest priority in the cycle to be aborted. This is exactly the function provided by the following algorithm.

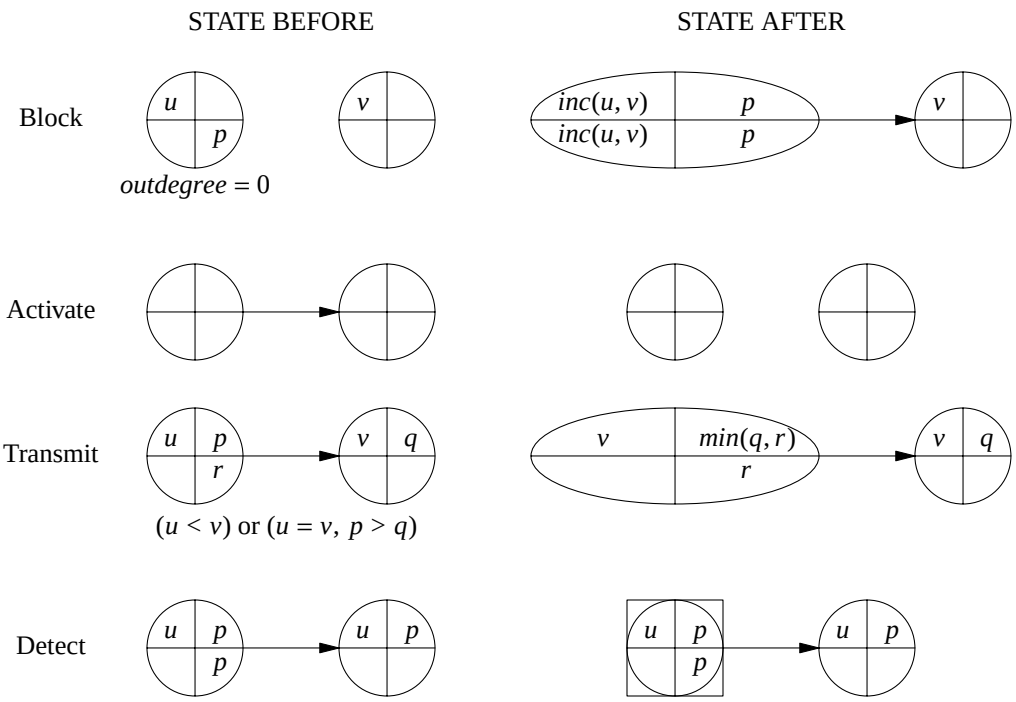


Figure 2

This algorithm is an extension of the last algorithm; unique public and private *priority numbers* have been added. In the figure above, public labels and priority numbers appear as the top left and right values in a node, and the private labels and priority numbers are the bottom left and right values, respectively. Initially, each process posts its unique private label and priority number. Nodes with equal public labels post the lower priority number (the Transmit step) and a node aborts when it sees its own private priority number together in a node with its own public label (the Detect step).

This algorithm aborts the lowest-priority process in any cycle that doesn't break on its own. Since the process with the highest public label may be waiting for a lock held by the lowest-priority process, the low-priority number may not start propagating around the cycle until after the public label has gone full-circle. Thus, this algorithm can take up to twice as long to detect deadlock as the first algorithm.

Theorem 2:
If a cycle of N nodes forms and persists long enough, the lowest priority process in the cycle will execute the Detect step after at least $N - 1$ *and at most* $2N - 2$ *consecutive Transmit steps.*

The proof of this theorem is similar to that of Theorem 1.

6. Discussion

In these algorithms, only one deadlocked process detects deadlock. This clears up the problem of deciding which process should initiate deadlock resolution. The detecting process could simply abort to break deadlock. In the first algorithm, an arbitrary process detects deadlock; and in the second, the process with lowest priority in the cycle will detect.

These algorithms do not use synchronized message passing. When one process posts a value in its public label, there is no guarantee that another process (executing the Transmit step) will read it before it is replaced by a new value. It is assumed that successive reads of a label will yield successive values; never out of order.

Process failures (in which all resources held are released) will not cause deadlocks to persist, but they may lead to false deadlock detection. This happens when the cycle has already been broken (by a time-out

or failure) when the deadlock is detected. Indeed, an actual cycle may never have existed in any single system state. If this is a problem in some applications, the deadlock-detecting process could initiate some type of cycle checking. Passing another set of messages around the cycle could at least guarantee that a real cycle existed at some point in time.

This paper assumes a system in which no central management of resources exists, but in which resources can be locked. Such a system has many advantages, but is open to livelock conditions.

The first algorithm has been implemented in an experimental database system and has performed well.

References

[BRACHA83] Bracha, Gabriel, Sam Toueg, "A Distributed Algorithm For Generalized Deadlock Detection," TR 83-558, June 1983, Department of Computer Science, Cornell University.

[CHANDY82] Chandy, K. M., J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1982, Ottawa, Canada.

[CHANG82] Change, Ernest J. H., "Echo Algorithms: Depth Parallel Operations on General Graphs," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982.

[DIJKSTRA80] Dijkstra, Edsger W., C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, Vol. 11, No. 1, August 1980.

[GLIGOR80] Gligor, Virgil and Susan H. Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, September 1980.

[MENASCE79] Menasce, Daniel and Richard Muntz, "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979.

[OBERMARCK82] Obermarck, Ron, "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems*, Vol. 7, No. 2, June 1982.