Deadlock Detection in Distributed Systems

Mukesh Singhal
Ohio State University

A distributed system is a network of sites that exchange information with each other by message passing. A site consists of computing and storage facilities and an interface to local users and to a communication network. A primary motivation for using distributed systems is the possibility of resource sharing — a process can request and release resources (local or remote) in an order not known a priori; a process can request some resources while holding others. In such an environment, if the sequence of resource allocation to processes is not controlled, a deadlock may occur.

A deadlock occurs when processes holding some resources request access to resources held by other processes in the same set. The simplest illustration of a deadlock consists of two processes, each holding a different resource in exclusive mode and each requesting an access to resources held by other processes. Unless the deadlock is resolved, all the processes involved are blocked indefinitely. Therefore, a deadlock requires the attention of a process outside those involved in the deadlock for its detection and resolution.

A deadlock is resolved by aborting one or more processes involved in the deadlock and granting the released resources to other processes involved in the deadlock. A process is aborted by withdrawing all its resource requests, restoring its state to an appropriate previous state, relinquishing all the resources it acquired after that state, and restoring all the relinquished resources to their original states. In the simplest case, a process is aborted by starting it afresh and relinquishing all the resources it held.

Deadlock is a constant problem, often offsetting the advantages of resource sharing. Deadlock handling is difficult in distributed systems because no site has accurate knowledge of the system state.

Resource vs. communication deadlock. Two types of deadlock have been discussed in the literature: resource deadlock and communication deadlock. In resource deadlocks, processes make access to resources (for example, data objects in database systems, buffers in store-and-forward communication networks). A process acquires a resource before accessing it and relinquishes it after using it. A process that requires resources for execution cannot proceed until it has acquired all those resources. A set of processes is resource-deadlocked if each process in the set requests a resource held by another process in the set.

In communication deadlocks, messages are the resources for which processes wait.[1] Reception of a message takes a process out of wait — that is, unblocks it. A set of processes is communication-deadlocked if each process in the set is waiting for a message from another process in the set and no process in the set ever sends a message. In this article we limit our discussion to resource deadlocks in distributed systems.

To present the state of the art of deadlock detection in distributed systems, this article describes a series of deadlock detection techniques based on centralized, hierarchical, and distributed control organizations. The article complements one by Knapp, which discusses deadlock detection in distributed database systems.[2] Knapp emphasizes the underlying theoretical principles of deadlock detection and gives an example of each principle. In contrast, this article examines deadlock detection in distributed systems more from the point of view of its practical implications. It presents an up-to-date and comprehensive survey of deadlock detection algorithms, discusses their merits and

0018-9162/89/1100-0037$01.00 © 1989 IEEE

drawbacks, and compares their performance (delays as well as message complexity). Moreover, this article examines related issues, such as correctness of the algorithms, performance of the algorithms, and deadlock resolution, which require further research.

Graph-theoretic model of deadlocks. The state of a system is in general dynamic; that is, system processes continuously acquire and release resources. Characterization of deadlocks requires a representation of the state of process-resource interactions. The state of process-resource interactions is modeled by a bipartite directed graph called a resource allocation graph. Nodes of this graph are processes and resources of a system, and edges of the graph depict assignments or pending requests. A pending request is represented by a request edge directed from the node of a requesting process to the node of the requested resource. A resource assignment is represented by an assignment edge directed from the node of an assigned resource to the node of the process assigned. For example, Figure 1 shows the resource allocation graph for two processes P_1 and P_2 and two resources R_1 and R_2, where edges $R_1 \rightarrow P_1$ and $R_2 \rightarrow P_2$ are assignment edges and edges $P_2 \rightarrow R_1$ and $P_1 \rightarrow R_2$ are request edges.

A system is deadlocked if its resource allocation graph contains a directed cycle in which each request edge is followed by an assignment edge. Since the resource allocation graph of Figure 1 contains a directed cycle, processes P_1 and P_2 are deadlocked. A deadlock can be detected by constructing the resource allocation graph and searching it for cycles.

In a distributed database system (DDBS), the user accesses the data objects of the database by executing transactions. A transaction can be viewed as a process that performs a sequence of reads and writes on data objects. The data objects of a database can be viewed as resources that are acquired (by locking) and released (by unlocking) by transactions. In DDBS literature the resource allocation graph is referred to as a transaction-wait-for (TWF) graph.[3] In a TWF graph, nodes are transactions and there is a directed edge from node T_1 to node T_2 if T_1 is blocked and must wait for T_2 to release some data object. A system is deadlocked if and only if there is a directed cycle in its TWF graph. Since both graphs denote the state of process-resource interaction, we will collectively refer to them as state graphs.

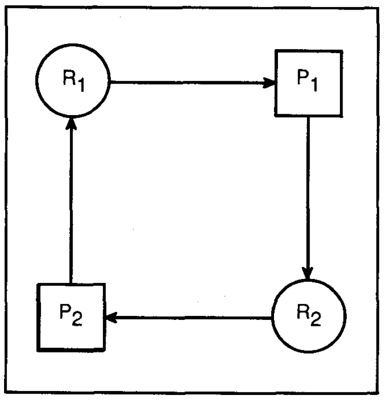

Figure 1. Resource allocation graph.

Deadlock-handling strategies

The three strategies for handling deadlocks are deadlock prevention, deadlock avoidance, and deadlock detection. In deadlock prevention, resources are granted to requesting processes in such a way that a request for a resource never leads to a deadlock. The simplest way to prevent a deadlock is to acquire all the needed resources before a process starts executing. In another method of deadlock prevention, a blocked process releases the resources requested by an active process.

In deadlock avoidance strategy, a resource is granted to a process only if the resulting state is safe. (A state is safe if there is at least one execution sequence that allows all processes to run to completion.) Finally, in deadlock detection strategy, resources are granted to a process without any check. Periodically (or whenever a request for a resource has to wait) the status of resource allocation and pending requests is examined to determine if a set of processes is deadlocked. This examination is performed by a deadlock detection algorithm. If a deadlock is discovered, the system recovers from it by aborting one or more deadlocked processes.

The suitability of a deadlock-handling strategy greatly depends on the application. Both deadlock prevention and deadlock avoidance are conservative, overly cautious strategies. They are preferred if deadlocks are frequent or if the occurrence of a deadlock is highly undesirable. In contrast, deadlock detection is a lazy, optimistic strategy, which grants a resource to a request if the resource is available, hoping that this will not lead to a deadlock.

Deadlock handling is complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every intersite communication involves a finite and unpredictable delay. Next, we examine the complexity and practicality of the three deadlock-handling approaches in distributed systems.

Deadlock prevention. Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process that holds the needed resource. In the former method, a process requests (or releases) a remote resource by sending a request message (or release message) to the site where the resource is located. This method has the following drawbacks:

(1) It is inefficient because it decreases system concurrency.

(2) A set of processes may get deadlocked in the resource-acquiring phase. For example, suppose process P_1 at site S_1 and process P_2 at site S_2 simultaneously request two resources R_3 and R_4 located at sites S_3 and S_4, respectively. It may happen that S_3 grants R_3 to P_1 and S_4 grants R_4 to P_2, resulting in a deadlock. This problem can be handled by forcing processes to acquire needed resources one by one, but that approach is highly inefficient and impractical.

(3) In many systems future resource requirements are unpredictable (not known a priori).

In the latter method, an active process forces a blocked process, which holds the needed resource, to abort. This method is inefficient because several processes may be aborted without any deadlock.

Deadlock avoidance. For deadlock avoidance in distributed systems, a resource is granted to a process if the resulting global system state is safe (the global state includes all the processes and resources of the distributed system). The following problems make deadlock avoidance impractical in distributed systems:

(1) Because every site has to keep track of the global state of the system, huge storage capacity and extensive communication ability are necessary.

(2) The process of checking for a safe global state must be mutually exclusive. Otherwise, if several sites concurrently perform checks for a safe global state (each site for a different resource request), they

may all find the state safe but the net global state may not be safe. This restriction severely limits the concurrency and throughput of the system.

(3) Due to the large numbers of processes and resources, checking for safe states is computationally expensive.

Deadlock detection. Deadlock detection requires examination of process-resource interactions for the presence of cyclic wait. In distributed systems deadlock detection has two advantages:

(1) Once a cycle is formed in the state graph, it persists until it is detected and broken.

(2) Cycle detection can proceed concurrently with the normal activities of a system; therefore, it does not have a serious effect on system throughput.

For these reasons the literature on deadlock handling in distributed systems is highly biased toward deadlock detection.

Issues in deadlock detection

Deadlock detection involves two basic tasks: maintenance of the state graph and search of the state graph for the presence of cycles. Because in distributed systems a cycle may involve several sites, the search for cycles greatly depends on how the system state graph is represented across the system.

Classified according to the way state graph information is maintained and the search for cycles is carried out, the three types of algorithms for deadlock detection in distributed systems are centralized, distributed, and hierarchical algorithms. In centralized algorithms the state graph is maintained at a single designated site, which has the sole responsibility of updating it and searching it for cycles. In distributed algorithms the state graph is distributed over many sites of the system, and a cycle may span state graphs located at several sites, making distributed processing necessary to detect it. In centralized algorithms the global state of the system is known and deadlock detection is simple. In distributed algorithms the problem of deadlock detection is more complex because no site may have accurate knowledge of the system state.[4] In hierarchical algorithms sites are arranged in a hierarchy, and a site detects deadlocks involving only its descendant sites. Hierarchical algorithms exploit access patterns local to a cluster of sites to efficiently detect deadlocks.

Correctness of deadlock detection algorithms. To be correct, a deadlock detection algorithm must satisfy two criteria:

(1) No undetected deadlocks: the algorithm must detect all existing deadlocks in finite time.

(2) No false deadlocks: the algorithm should not report nonexistent deadlocks.

In distributed systems where there is no global memory and communication occurs solely by messages, it is difficult to design a correct deadlock detection algorithm because sites may receive out-of-date and inconsistent state graphs of the system. As a result, sites may detect a cycle that never existed but whose different segments existed in the system at different times. That is why many deadlock detection algorithms reported in the literature are incorrect.

Strengths and weaknesses of centralized algorithms. In centralized deadlock detection algorithms, a designated site, often called the control site, has the responsibility of constructing the global state graph and searching it for cycles. The control site may maintain the global state graph all the time, or it may build it whenever deadlock detection is to be carried out by soliciting the local state graph from every site. Centralized algorithms are conceptually simple and are easy to implement. Deadlock resolution is simple in these algorithms — the control site has the complete information about the deadlock cycle, and it can optimally resolve the deadlock.

However, because control is centralized at a single site, centralized deadlock detection algorithms have a single point of failure. Communication links near the control site are likely to be congested because the control site receives state graph information from all the other sites. Also, the message traffic generated by deadlock detection activity is independent of the rate of deadlock formation and the structure of deadlock cycles.

Strengths and weaknesses of distributed algorithms. In distributed deadlock detection algorithms, the responsibility of detecting a global deadlock is shared equally among the sites. The global state graph is spread over many sites, and several sites participate in the detection of a global cycle. Unlike centralized algorithms, distributed algorithms are not vulnerable to a single point of failure, and no site is swamped with deadlock detection activity. Deadlock detection is initiated only if a waiting process is suspected to be part of a deadlock cycle.

But deadlock resolution is often cumbersome in distributed deadlock detection algorithms because several sites may detect the same deadlock and may not be aware of other sites and/or processes involved in the deadlock. Distributed algorithms are difficult to design because sites may collectively report the existence of a global cycle after seeing its segments at different instants (though all the segments never existed simultaneously) due to the system's lack of globally shared memory. Also, proof of correctness is difficult for these algorithms.

Strengths and weaknesses of hierarchical algorithms. In hierarchical deadlock detection algorithms, sites are arranged hierarchically, and a site detects deadlocks involving only its descendant sites. To efficiently detect deadlocks, hierarchical algorithms exploit access patterns local to a cluster of sites. They tend to get the best of both worlds: they have no single point of failure (as centralized algorithms have), and a site is not bogged down by deadlock detection activities that it is not concerned with (as sometimes happens in distributed algorithms). For efficiency, most deadlocks should be localized to as few clusters as possible; the objective of hierarchical algorithms will be defeated if most deadlocks span several clusters.

Next, we describe a series of centralized, distributed, and hierarchical deadlock detection algorithms. We discuss the basic idea behind their operations, compare them with each other, and discuss their pros and cons. We also summarize the performance of these algorithms in terms of message traffic, message size, and delay in detecting a deadlock (see Table 1). It is not possible to enumerate these performance measures with high accuracy for many deadlock detection algorithms for the following reasons: the random nature of the TWF graph topology, the invocation of deadlock detection activities even though there is no deadlock, and the initiation of deadlock detection by several processes in a deadlock cycle. Therefore, for most algorithms we have given performance bounds rather than exact numbers (for example, the maximum number of messages transferred to detect a global cycle).

Table 1. **Performance comparison of distributed deadlock detection algorithms.**

Algorithm	Number of Messages	Delay	Message Size
Goldman	$<m.n$	$\tau + n\mathrm{T}$	Variable (medium)
Isloor-Marsland	$r(N-1)$	0	Constant (small)
Menasce-Muntz	$m(n-1)$	$n\mathrm{T}$	Variable (small)
Obermarck	$m(n-1)/2$	$n\mathrm{T}$	Variable (medium)
Chandy et al.	$<m.n$	$\tau + n\mathrm{T}$	Constant (small)
Haas-Mohan	$m(n-1)$	$\tau + (n-1)\mathrm{T}$	Variable (medium)
Sugihara et al.	$<m.n$	$(n-1)\mathrm{T}$	Constant (small)
Sinha-Natarajan	best confi. $2(n-1)$ worst confi. $m(n-1)$	$2(n-1)\mathrm{T}$	Constant (small)
Mitchell-Merritt	$m(n-1)$	$(n-1)\mathrm{T}$	Constant (small)
Bracha-Toueg	$4m(N-1)$	$4d\mathrm{T}$	Variable (medium)

N: number of sites; n: number of sites in deadlock cycle;
m: number of processes involved in deadlock;
T: intersite communication delays;
τ: deadlock initiation delay;
r: TWF graph update rate; d: diameter of TWF graph

Centralized deadlock detection algorithms

In the simplest centralized deadlock detection algorithm, a designated site called the control site maintains the state graph of the entire system and checks it for the existence of deadlock cycles. All sites request or release resources (even local resources) by sending "request resource" or "release resource" messages to the control site. When the control site receives such a message, it correspondingly updates its state graph. This algorithm is conceptually simple and easy to implement. However, it is highly inefficient because all the resource acquisition and release requests must go through the control site, causing large delays in response to user requests, large communication overhead, and congestion of communication links near the control site. Also, the algorithm's reliability is poor because if the control site fails, not only does deadlock detection stop, but also the entire system comes to a halt. This is because all the status information resides at the control site.

Some problems of this algorithm (such as long response time and congested communication links near the control site) can be mitigated by having each site maintain its resource status (state graph) locally and by having every site send its resource status to a designated site periodically for construction of the global state graph (and detection of deadlocks). However, due to the inherent communication delay and the lack of perfectly synchronized clocks in distributed systems, the designated site may get an inconsistent view of the system and detect false deadlocks.[5]

For example, suppose two resources R_1 and R_2 are stored at sites S_1 and S_2, respectively. Suppose the following two transactions T_1 and T_2 are started almost simultaneously at sites S_3 and S_4, respectively:

T_1	T_2
Lock R_1	Lock R_1
Unlock R_1	Unlock R_1
Lock R_2	Lock R_2
Unlock R_2	Unlock R_2

Suppose the Lock R_1 request of T_1 arrives at S_1 and locks R_1, followed by the Lock R_1 request of T_2, which waits at S_1. At this point S_1 reports its status, $T_2 \rightarrow T_1$, to the designated site. Suppose the Lock R_2 request of T_2 arrives at S_2 and locks R_2, followed by the Lock R_2 request of T_1, which waits at S_2. At this point S_2 reports its status, $T_1 \rightarrow T_2$, to the designated site, which, after constructing the global state graph, reports a false deadlock $T_1 \rightarrow T_2 \rightarrow T_1$.

Ho-Ramamoorthy algorithm. In an attempt to solve the problem of the above algorithm, Ho and Ramamoorthy have presented two centralized deadlock detection algorithms called the two-phase and one-phase algorithms.[5] These algorithms, respectively, collect two consecutive status reports and keep two status tables at each site to ensure that the control site gets a consistent view of the system.

Two-phase algorithm. Every site maintains a status table, which contains the status of all processes initiated at that site. The status of a process includes all resources locked and all resources being waited for. Periodically, a designated site requests the status table from all sites, constructs a state graph from the received information, and searches it for cycles. If there is no cycle, then the system is not deadlocked. Otherwise, the designated site again requests status tables from all the sites and again constructs a state graph, using only the transactions common to both reports. If the same cycle is detected again, the system is declared deadlocked.

Ho and Ramamoorthy claimed that by selecting the transactions common to two consecutive reports, the designated site gets a consistent view of the system (a view that correctly reflects the state of the system) because if a deadlock exists, the same wait-for condition must exist in both reports. However, this claim is incorrect — a cycle in the wait-for conditions of the transactions common to two consecutive reports does not imply a deadlock — and the two-phase algorithm can report false deadlocks. By getting two consecutive reports, the designated site reduces but does not eliminate the probability of getting an inconsistent view.

One-phase algorithm. The one-phase algorithm requires only one phase of status reports from sites, but each site maintains two status tables: a resource status table and a process status table. The resource status table keeps track of the transactions that have locked or are waiting for resources stored at that site. The process status table keeps track of the resources locked by or waited for by all the transactions at that site. Periodically, a designated site requests both tables from all other sites, constructs a state graph using only the transactions for which the resource table matches the process table, and searches it for cycles. If no cycle is found, then the system is not deadlocked; otherwise, a deadlock has been detected.

The algorithm does not detect false deadlocks because it eliminates inconsistency in state information by using only the information common to both tables. For example, if the resource table at S_1 indicates that resource R_1 is being waited for by a process P_2 (i.e., $R_1 \leftarrow P_2$) and the process table at S_2 indicates that process P_2 is waiting for resource R_1 (i.e., $P_2 \rightarrow R_1$), then edge $P_2 \rightarrow R_1$ in the constructed state graph correctly reflects the system state. If either of these entries is missing from the resource or process table, then a request message or release message from S_2 to S_1 is in transit and $P_2 \rightarrow R_1$ cannot be ascertained. The one-phase algorithm is faster and requires fewer messages than the two-phase algorithm. But it requires more storage because every site maintains and exchanges two status tables.

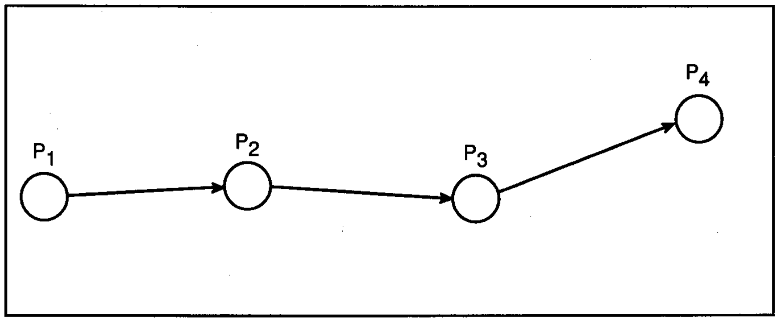

Figure 2. Example of OBPL (ordered blocked process list).

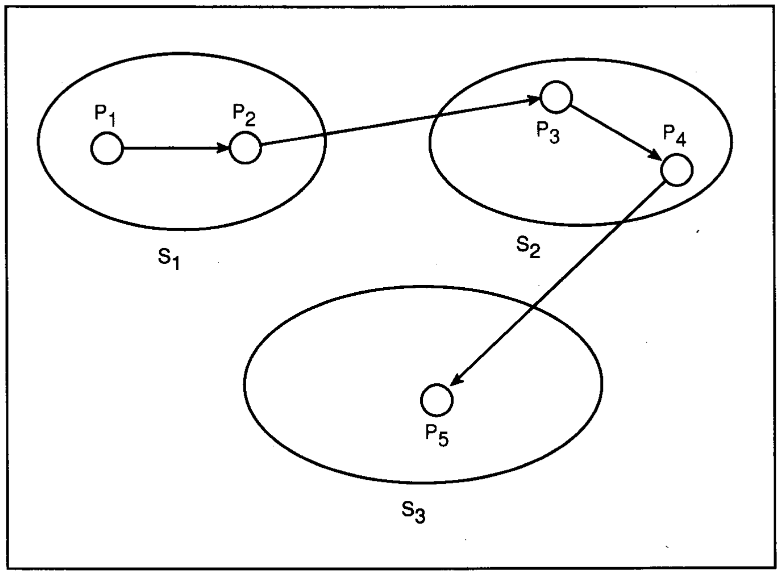

Figure 3. Example of Goldman's algorithm.

Distributed deadlock detection algorithms

In distributed algorithms all sites cooperate to detect a cycle in the state graph, which is distributed over several sites of the system. Deadlock detection can be initiated whenever a process is forced to wait, and it can be initiated either by the local site of the process or by the site where the process waits. Information about the state graph can be maintained and circulated in various forms (for example, table,[5] list,[6] string,[7,8] and probe[1,9]) during the deadlock detection phase.

Goldman's algorithm. Goldman's algorithm[6] exchanges deadlock-related information in the form of an ordered blocked process list (OBPL), in which each process (except the last) is blocked by its successor. The last process in an OBPL may either be waiting to access a resource or be running. For example, OBPL P_1, P_2, P_3, P_4 represents the state graph in Figure 2.

The algorithm detects a deadlock by repeatedly expanding the OBPL, appending the process that holds the resource needed by the last process in the list until either a deadlock is discovered (that is, the last process is blocked by a process in the list) or the OBPL is discarded (the last process is running). As an example, suppose in the system shown in Figure 3 process P_1 initiates deadlock detection and sends OBPL P_1, P_2 to process P_2. When process P_2 receives the OBPL, it appends P_3 to the OBPL and sends the new OBPL P_1, P_2, P_3 to P_3. Likewise, P_3 sends OBPL P_1, P_2, P_3, P_4 to P_4 and P_4 sends OBPL P_1, P_2, P_3, P_4, P_5 to P_5. When P_5 receives the OBPL, it discards the OBPL because it is not blocked. Had P_5 been blocked by P_1, P_2, P_3, or P_4, a deadlock would have been detected by P_5.

An advantage of Goldman's algorithm is that it does not require continuous maintenance of TWF graphs. It constructs an OBPL whenever deadlock detection is to be carried out. However, it requires that every process have at most one outstanding resource request.

Isloor-Marsland algorithm. The "on-line" deadlock detection algorithm of Isloor and Marsland[10] detects deadlocks at the earliest possible instant — that is, at the time of making decisions about data allocation at the concerned site. It is based on the concept of reachable set. The reachable set of a node in the state graph is the set of all the nodes that can be reached from it. A process is deadlocked if the reachable set of the corresponding node contains the node itself.

The algorithm detects deadlocks by constructing reachable sets and checking whether any node belongs to its own reachable set. To do this, every site maintains the system state graph and reachable sets for each node in the state graph; the reachable sets are continually updated whenever edges are added to or deleted from the state graph. Whenever a resource is allocated,

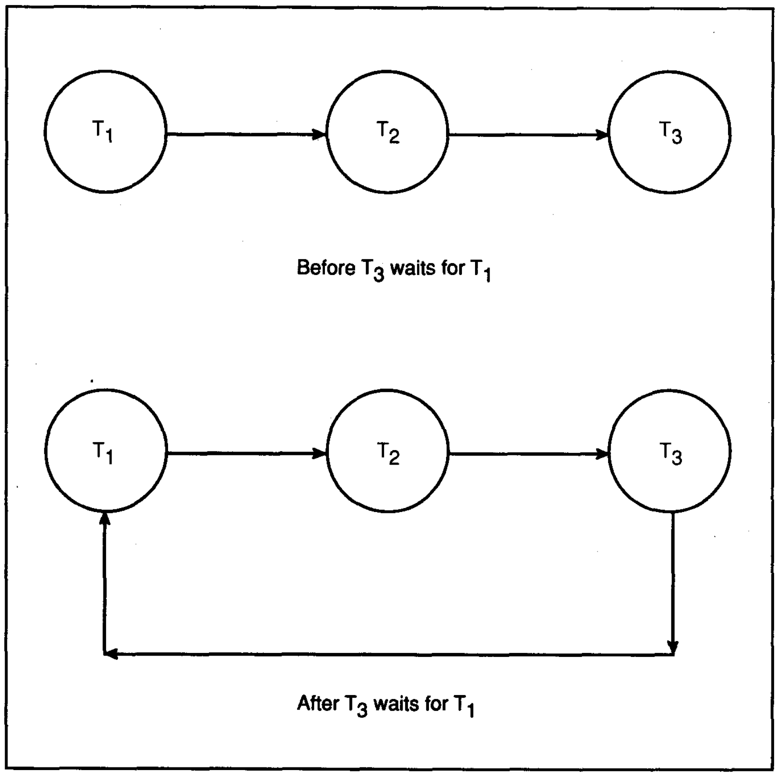

Figure 4. Example of Menasce-Muntz algorithm.

whenever a process is made to wait for a resource, or whenever a process releases a resource, the corresponding information is broadcast to all other sites. Therefore, if r changes per second occur in the state graph, then the algorithm requires $r(N\text{-}1)$ messages per second for deadlock detection. However, the messages are short because they contain only an update to the state graph resulting from the execution of a request.

Menasce-Muntz algorithm. The deadlock detection algorithm of Menasce and Muntz[3] propagates only the two end points of a directed path (called a blocking pair), rather than the whole path, to detect deadlocks. The blockingset(T) of a transaction T is the set of all nonblocked transactions that can be reached from T by following all directed paths in the TWF graph. This is the set of transactions responsible for blocking the transaction T. When a transaction T gets blocked, then for each transaction T_i in the blockingset(T), the algorithm sends the blocking pair (T, T_i) to the home sites of T and T_i. (In other words, information about the condensed TWF graph is sent along the paths of the global TWF graph.)

Figure 4 illustrates the algorithm for a deadlock involving three transactions T_1, T_2, and T_3. Initially T_1 is blocked by T_2 and T_2 by T_3, and the home sites of T_1 and T_2 have the knowledge of the TWF graph $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$, respectively. Now, when T_3 makes a request and is blocked by T_1, the blocking pair (T_3, T_2) is sent to the home site of T_2. This causes an edge from T_3 to T_2 to be added in the TWF of the home site of T_2, resulting in a cycle $T_3 \rightarrow T_2 \rightarrow T_3$ and detection of a deadlock at the home site of T_2.

Gligor and Shattuck[4] have shown that this algorithm fails to detect some deadlocks for two reasons: First, in the case of a nonlocal request, the determination of whether a transaction is blocked or not is incorrect because that determination cannot be made until the response arrives from a remote site. Second, even if this response arrives to determine correctly whether a transaction is blocked or not, the algorithm does not make use of it. Gligor and Shattuck have fixed this algorithm by precisely defining the status of all the transactions, whether active, blocked, or waiting for the outcome of a nonlocal resource request, and by having the algorithm propagate appropriate blocking pairs when it becomes certain that a waiting transaction is blocked.

Obermarck's algorithm. In Obermarck's algorithm,[8] the nonlocal portion of the global TWF graph at a site is abstracted by a distinct node, called "external" or Ex, which helps determine potential multisite deadlocks without requiring a huge global TWF graph to be stored at each site. Deadlock detection at a site follows the following iterative process:

(1) A site waits for deadlock-related information (produced in the previous deadlock detection iteration) from other sites.

(2) The site combines the received information with its local TWF graph, detects all cycles, and breaks only cycles that do not contain the node Ex.

(3) For all cycles $Ex \rightarrow T_1 \rightarrow T_2 \rightarrow Ex$ that contain the node Ex (these cycles are potential candidates for global deadlocks), the site transmits them in string form Ex, T_1, T_2, Ex to all other sites.

The algorithm reduces message traffic by lexically ordering the nodes (transactions) and sending a string Ex, T_1, T_2, T_3, Ex to other sites only if T_1 is higher than T_3 in the lexical ordering.

Chandy-Misra-Haas algorithm. Chandy, Misra, and Haas's algorithm[1] uses a special message called a probe. A probe is a triplet (i, j, k) denoting that it belongs to a deadlock detection initiated for process P_i and is being sent by the home site of P_j to the home site of P_k. A probe message travels along the edges of the global TWF graph, and a deadlock is discovered when a probe message returns to its initiating process. As an example, consider the system shown in Figure 5. If process P_1 initiates deadlock detection, it sends probe (1, 3, 4) to the controller C_2 at site S_2. Since P_6 is waiting for P_8 and P_7 is waiting for P_{10}, C_2 sends probes (1, 6, 8) and (1, 7, 10) to C_3, which in turn sends probe (1, 9, 1) to C_1. On receipt of probe (1, 9, 1), C_1 declares that P_1 is deadlocked.

In Haas and Mohan's algorithm,[7] a variation of Chandy, Misra, and Haas's algorithm, a process comes to know (besides detecting that it is deadlocked) all the deadlock cycles in which it is involved. The algorithm achieves this by passing more information about the potential cycles in a probe message. In this algorithm a message consists of not only the information about the initiator of the deadlock, but also all the paths to it.

Mitchell-Merritt algorithm. In the deadlock detection algorithm of Mitchell and Merritt,[11] each node of the TWF graph has two labels: private and public. The private label of each node is unique to that node, and initially both labels have the same value. The algorithm detects a deadlock by propagating the public labels of nodes in a backward direction in the TWF graph. When a transaction gets blocked, the public and private labels of its node in the TWF graph are changed to a value greater than their previous values and greater than the public labels of the blocking transaction. (Blocked transactions update their labels in this manner periodically.) A deadlock is detected when a transaction receives its own public label. In essence, the largest public label propagates in a backward direction in a deadlock cycle. Deadlock resolution is simple in this algorithm because only one process detects a deadlock (that process can resolve the deadlock by aborting itself).

Sugihara et al.'s algorithm. As in Obermarck's algorithm, in Sugihara et al.'s algorithm[12] every site maintains only a local TWF graph. A wait for a remote resource is reflected by adding a global edge to the local TWF graphs of the site of the requesting process and the site of the process holding the requested resource. In a global edge the nodes corresponding to the requesting process and the process holding the requested resource are referred to as the O-node and the I-node, respectively (Figure 6).

A site initiates deadlock detection by sending a message, similar to the probe message of Chandy, Misra, and Haas, whenever the addition of an edge (due to a resource wait) in its TWF graph creates a new path between any of its I-nodes and O-nodes. Note that a site can be involved in a global deadlock only when there is a path between some of its I-nodes and O-nodes. The algorithm has a unique resolver for every deadlock, and deadlock resolution does not cause detection of false deadlocks. An advantage of the algorithm is that a site maintains minimal information about the global TWF graph.

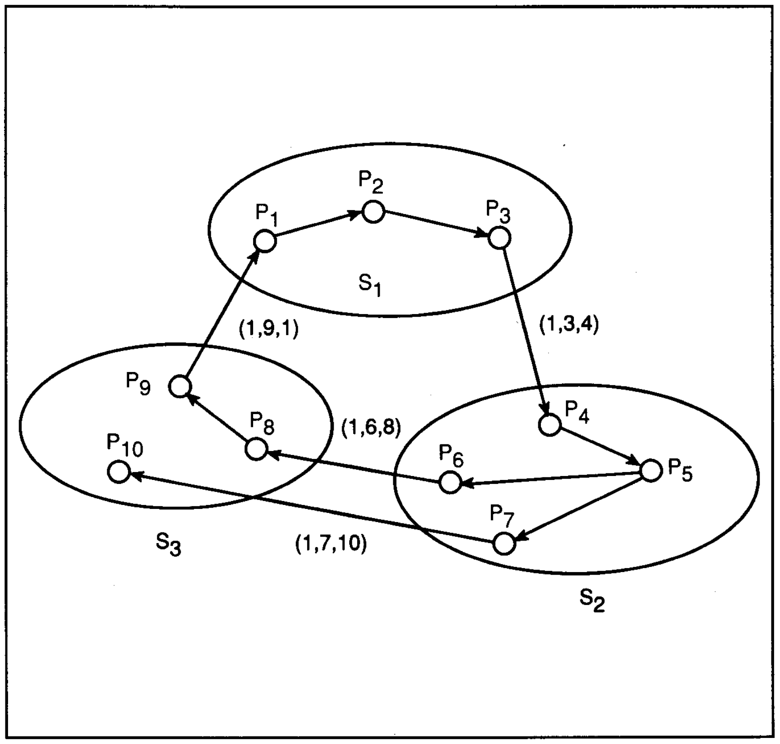

Figure 5. Example of Chandy-Misra-Haas algorithm.

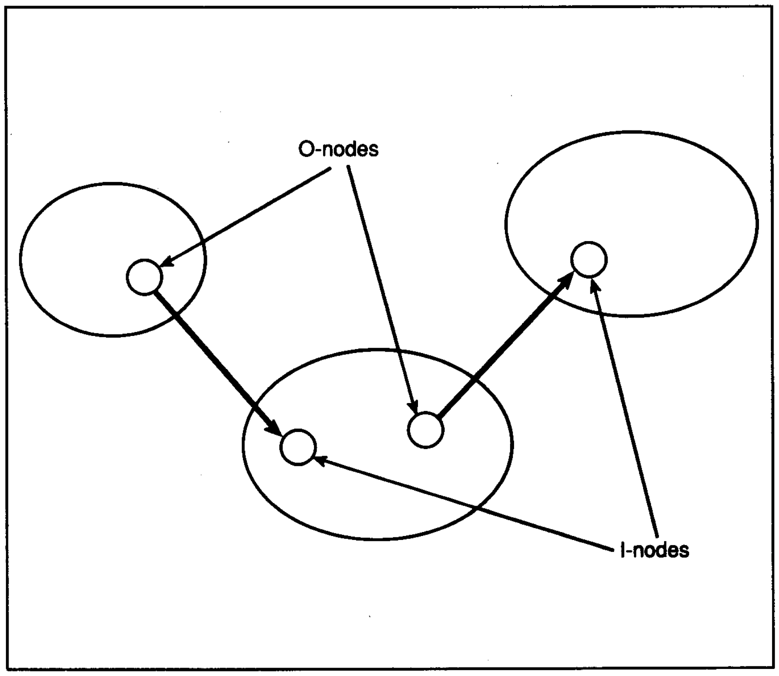

Figure 6. I-nodes and O-nodes.

Sinha-Natarajan algorithm. Sinha and Natarajan's algorithm[9] does not construct the TWF graph, but it follows the edges of the graph to search for cycles. Transactions are prioritized, and an antagonistic conflict is said to occur when a transaction waits for a data object that is locked by a lower-priority transaction. The algorithm initiates deadlock detection only when an antagonistic conflict occurs, rather than whenever a transaction begins to wait for another transaction. Therefore, it requires fewer messages to detect deadlocks and generates fewer messages during normal conditions.

The algorithm detects a deadlock by circulating a probe message through a cycle in the global TWF graph. A probe message is a 2-tuple (i, j) where i is the transaction that faced the antagonistic conflict and initiated deadlock detection and j is the transaction of the lowest priority among all the transactions (nodes of the TWF graph) the probe has traversed so far. When a waiting transaction receives a probe initiated by a lower-priority transaction, the probe is discarded. (Thus, the algorithm filters out redundant messages).

An interesting property of this algorithm is that a deadlock is detected when the probe issued by the highest-priority process in the cycle returns to that process. (There is only one detector of every deadlock.) Deadlock resolution is simple; the detector of a deadlock can resolve the deadlock by aborting the lowest-priority transaction of the cycle. Choudhary et al. have shown that this algorithm detects false deadlocks and fails to report all deadlocks because it overlooks the possibility of a transaction waiting transitively on a deadlock cycle and because probes of aborted transactions are not deleted properly.[13]

Badal's algorithm. Badal's algorithm[14] exploits the fact that deadlocks can be divided into several categories based on the complexity of their topology; the frequency of deadlock occurrence and the costs of deadlock detection differ among categories. There is no point in detecting simple deadlocks with an algorithm designed to detect complex deadlocks. Badal optimizes performance by using three levels of deadlock detection; activity at each level is more complex (and expensive) than at the preceding level. Deadlock detection starts at the first level algorithm and is delegated to the next higher level if the current-level algorithm fails to report a deadlock. The third-level algorithm is designed to detect global deadlocks that escape the first two levels, and it closely resembles Obermarck's algorithm.

The most attractive feature of Badal's algorithm is that it detects the most frequent deadlocks with minimum overhead (first- and second-level algorithms) and switches to an expensive algorithm (third level) only when really needed. However, it has a fixed overhead due to information kept in lock tables, frequent checking of deadlocks of length two, and longer messages. Consequently, it is most suitable in environments where deadlocks occur frequently, justifying the fixed overhead.

Bracha-Toueg algorithm. In Bracha and Toueg's deadlock detection algorithm for generalized environments, called the r-out-of-s request model,[15] a process can request any r resources from a pool of s resources. After issuing an r-out-of-s request, a process remains blocked until it gets any r out of the s resources. The algorithm consists of two phases: notify and grant. In the first phase, notify messages are propagated downward in forestlike patterns of the TWF graph; in the second phase, grant messages are echoed back from all active processes, simulating the granting of resources to requests. At the end of the second phase, all the processes that are not made active are deadlocked.

Because the system state is dynamic, the TWF graph may change during the execution of the algorithm. The algorithm overcomes such changes by propagating special Freeze messages throughout the system. When a process receives a Freeze message, it saves a snapshot of its state. Deadlocks are detected by running the deadlock detection algorithm on the collection of snapshots thus obtained.

Hierarchical deadlock detection algorithms

In hierarchical algorithms sites are (logically) arranged in hierarchical fashion, and a site is responsible for detecting deadlocks involving only its children sites. To optimize performance, these algorithms take advantage of access patterns localized to a cluster of sites.

Menasce-Muntz algorithm. In the hierarchical deadlock detection algorithm of Menasce and Muntz,[3] all the resource controllers are arranged in tree fashion. The controllers at the bottommost level, called leaf controllers, manage resources; the others, called nonleaf controllers, are responsible for deadlock detection. A leaf controller maintains the part of the global TWF graph that is concerned with the allocation of the resources at that leaf controller. A nonleaf controller maintains the TWF graph spanning its children controllers and is responsible for detecting only deadlocks involving its own leaf controllers. Whenever a change occurs in a controller's TWF graph due to a resource allocation, wait, or release, the change is propagated to its parent controller. The parent controller makes the changes in its TWF graph, searches for cycles, and propagates the changes upward if necessary. A nonleaf controller can be kept up to date about the TWF graphs of its children continuously (that is, whenever a change occurs) or periodically.

Ho-Ramamoorthy algorithm. In the hierarchical algorithm of Ho and Ramamoorthy,[5] sites are grouped into several disjoint clusters. Periodically, a site is chosen as the central control site, which dynamically chooses a control site for each cluster (Figure 7). The central site requests every control site for its intercluster transaction status information and wait-for relations. As a result, a control site collects status tables from all the sites in its cluster and applies the one-phase deadlock detection algorithm to detect all deadlocks involving only intracluster transactions. Then it sends intercluster transaction status information and wait-for relations (derived from the information thus collected) to the central site. The central site splices the intercluster information thus received, constructs a system state graph, and searches it for cycles. Thus, a control site detects all deadlocks located in its cluster, and the central site detects all intercluster deadlocks.

Future research directions

Several issues related to deadlock detection in distributed systems have not been adequately studied and require further research.

Algorithm correctness. There is a dearth of sophisticated formal methods to prove the correctness of deadlock detection algorithms for distributed systems.

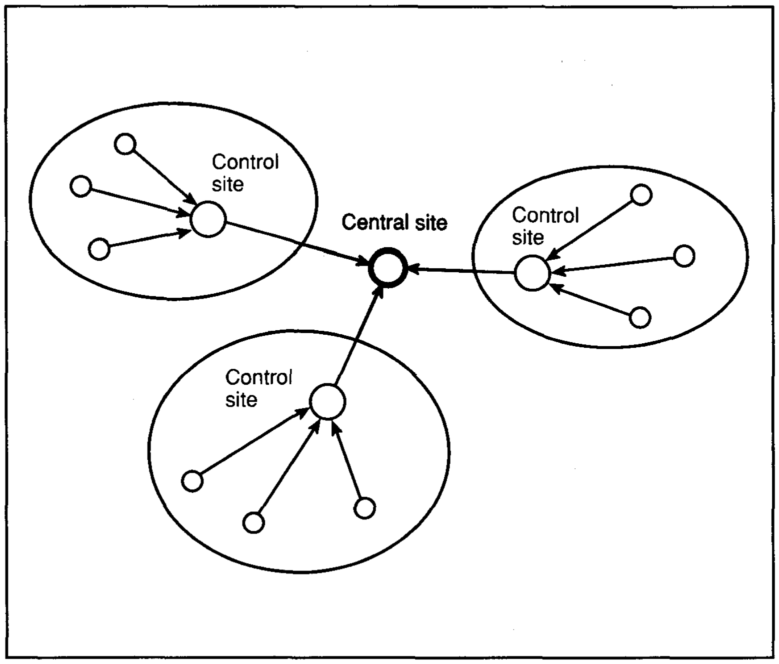

Figure 7. Example of Ho-Ramamoorthy algorithm.

Most researchers have used informal, intuitive arguments to show the correctness of their algorithms. But intuition has proved to be highly unreliable, and more than half the algorithms have been found incorrect. A formal proof of the correctness of deadlock detection algorithms is difficult for several reasons:

(1) The TWF graph and deadlock cycles can form in innumerable ways, making it difficult to imagine and exhaustively study all conceivable situations.

(2) Deadlock is very sensitive to the timing of requests.

(3) In distributed systems, message delays are unpredictable and there is no global memory.

Time-dependent proof techniques are particularly necessary.

Algorithm performance. Although many deadlock detection algorithms have been proposed for distributed systems, their performance analysis has not received sufficient attention. Most authors (for example, Obermarck and Sinha and Natarajan) have evaluated their algorithms on the basis of the number of messages exchanged to detect an existing cycle in the TWF graph. This performance criterion is deceptive because deadlock detection algorithms also exchange messages during normal conditions (when there is no deadlock). The number of messages exchanged may not be a true indicator of communication overhead because some algorithms[7,8,10] exchange long messages whereas others[1,3] exchange short messages. Therefore, we require a different criterion for computing communication overhead, which should take into account the number as well as the size of messages exchanged, not only in deadlocked conditions but also in normal conditions.

The persistence of deadlocks results in wasteful utilization of resources and increased response time to user requests. Therefore, an important performance measure of deadlock detection algorithms is the average deadlock persistence time. There is often a trade-off between message traffic and deadlock persistence time. For example, although the on-line deadlock detection algorithm of Isloor and Marsland detects a deadlock at the earliest instant, it has high message traffic. On the other hand, Obermarck's algorithm has less message traffic, but its deadlock persistence time is proportional to the size of the cycle. This trade-off is intuitive — quick detection of deadlock requires fast dissemination of information about the state graph, which implies high message traffic.

There is another dimension to the trade-off between message traffic and deadlock persistence time. For example, Chandy, Misra, and Haas's algorithm requires short messages, but when it detects a deadlock, it takes a while to resolve it. In contrast, Haas and Mohan's algorithm exchanges longer messages; however, when a process detects a deadlock, it knows all the processes involved in it, and therefore the deadlock can be resolved quickly.

Besides communication overhead and deadlock persistence time, any evaluation of deadlock detection algorithms should consider measures such as storage overhead for deadlock detection information, processing overhead to search for cycles, and additional processing overhead to optimally resolve deadlocks. Among the factors that influence these measures are the techniques used for deadlock detection, the data access behavior of processes, the request-release pattern of processes, and resource holding time. How these factors influence performance and how the performance characteristics of different detection algorithms compare with each other are not well understood. A complete performance study of deadlock detection algorithms calls for the development of performance models, the measurement of performance using analytic or simulation techniques, and a performance comparison of existing algorithms.

Deadlock resolution. Persistence of a deadlock has two major disadvantages: First, all the resources held by deadlocked processes are not available to any other process. Second, the deadlock persistence time gets added to the response time of each process involved in the deadlock. Therefore, the problem of promptly and efficiently resolving a detected deadlock is as important as the problem of deadlock detection itself. Unfortunately, most deadlock detection algorithms for distributed systems do not address the problem of deadlock resolution.

A deadlock is resolved by aborting at least one process involved in the deadlock and granting the released resources to other processes involved in the deadlock. Efficient resolution of a deadlock requires knowledge of all the processes involved in the deadlock and all resources held by these processes. When a deadlock is detected, the speed of its resolution depends on how much information about it is available, which in turn depends on how much

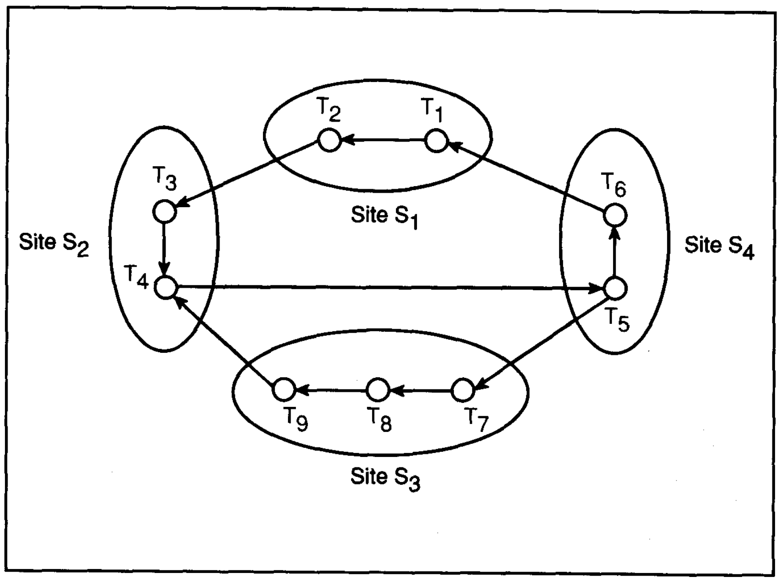

Figure 8. Detection of a false deadlock.

information is passed around during the deadlock detection phase. In existing distributed deadlock detection algorithms, deadlock resolution is complicated by at least one of the following problems:

• A process that detects a deadlock does not know all the processes (and resources held by them) involved in the deadlock — for example, the algorithms of Chandy, Misra, and Haas and Menasce and Muntz.

• Two or more processes may independently detect the same deadlock — for example, the Chandry-Misra-Haas and Goldman algorithms. If every process that detects a deadlock resolves it, then deadlock resolution will be inefficient because several processes will be aborted to resolve a deadlock (different processes may choose to abort different processes). Therefore, we need some postdetection processing to select a process to be responsible for resolving the deadlock.

Many deadlock detection algorithms require an additional round of message exchanges to select a deadlock resolver and/or to gather the information needed to efficiently resolve a deadlock. The Sinha-Natarajan algorithm is one of the exceptions where each deadlock is detected only by the highest-priority process that (upon deadlock detection) knows the lowest-priority processes in the deadlock cycle. There is often a trade-off between the volume of information exchanged during the deadlock detection phase and the amount of time needed to resolve a deadlock once it is detected. For example, Haas and Mohan's algorithm exchanges long messages, but when a deadlock is detected, its resolution is quick. On the other hand, in Menasce and Muntz's algorithm the messages exchanged are short, but when a deadlock is detected, its resolution is tedious and time consuming.

Whether it is better to exchange long messages during the deadlock detection phase and resolve a detected deadlock quickly or to exchange short messages during the deadlock detection phase and do extra computation to resolve a detected deadlock depends on how frequently deadlocks occur in a system. If deadlocks are frequent, then the former approach should perform better and vice versa. Even after all the information necessary to resolve a deadlock is available, resolution involves the following nontrivial steps:

(1) Select a victim (the process to be aborted) for the optimal resolution of a deadlock (this step may be computationally tedious).

(2) Abort the victim, release all the resources held by it, restore all the released resources to their previous states, and grant the released resources to deadlocked processes.

(3) Delete all the deadlock detection information concerning the victim at all sites.

Execution of the second step is complicated in environments where a process can simultaneously wait for multiple resources because the allocation of a released resource to another process can cause a deadlock. The third step is even more critical because if the information about the victim is not deleted quickly and properly, it may be counted in several other (false) cycles, causing detection of false deadlocks. As Choudhary et al. point out, the failure to delete probe messages in the Sinha-Natarajan algorithm causes the detection of false deadlocks. To be safe, during the execution of the second and third steps, the deadlock detection process (at least in potential deadlocks that include the victim) must be halted to avoid detection of false deadlocks. In the Sugihara et al. algorithm, a control token serializes global deadlock resolution to eliminate its side effects on the deadlock detection process.

False deadlocks. In environments where a process can simultaneously wait for multiple resources, deadlock resolution is even more complex because an edge may be shared by two or more cycles, and deleting that edge will break all those deadlocks. However, since the search for each cycle is carried out independently, deadlock detection initiated for some cycles may not be aware of the deleted edge, resulting in detection of false deadlocks. Figure 8 illustrates such a scenario. Two deadlocks share an edge (T_4, T_5). Suppose the top cycle has been detected by process T_3, which is breaking it by deleting the edge (T_4, T_5). Concurrently process T_8 may initiate a deadlock detection message, and it may happen that T_3 breaks the edge (T_4, T_5) after the deadlock detection message initiated by T_8 has crossed (or traversed) it. In this case, T_8 will detect a (false) deadlock involving processes T_4, T_5, T_7, T_8, and T_9, which has already been broken by T_3.

In brief, deadlock detection involves detecting a static condition — once a deadlock cycle is formed, it persists until it is detected and broken. On the other hand, deadlock resolution is a dynamic process — it changes the state graph by deleting its edges and nodes. Two forces are working in opposite directions: the wait for resources adds edges and nodes to the state graph, while deadlock resolution removes them from the state graph. Therefore, if deadlock resolution is not carefully incorporated into deadlock detection, false deadlocks are likely to be detected.

Deadlock probability. The frequency of deadlocks is a crucial factor in the design of distributed systems. If deadlocks are infrequent, then a time-out mechanism is the best approach to handling deadlocks because it has very low overhead. In a time-out mechanism a transaction or a process is aborted after it has waited for more than a specified period, called the time-out interval, after issuing a resource request. The most critical issue in a time-out mechanism is to choose an appropriate time-out interval; if the time-out interval is short, then many transactions may be aborted unnecessarily (that is, without being deadlocked), and if the time-out interval is long, then deadlocks will persist for a long time. The time-out mechanism is also susceptible to cyclic restarts, in which transactions are repeatedly aborted and restarted.

The probability of deadlocks depends on factors such as process mix, resource request and release patterns, resource holding time, and the average number of data objects held (locked) by processes. The probability of deadlocks is difficult to analyze because deadlock occurrence is highly sensitive to the timing and order in which resource requests are made. Gray et al.[16] have done an approximate analysis of deadlock probability and found that

(1) transaction waits and deadlocks are rare, but they both increase linearly with the degree of multiprogramming,
(2) most deadlocks are of length (size) two,
(3) deadlocks rise as the fourth power of transaction size, and
(4) waits rise as the second power of transaction size.

The probability of deadlock is an important parameter, and any further work in this direction will be a worthwhile contribution.

To sum up, of the three types of algorithms for detecting global deadlocks, distributed are the most prominent and most thoroughly investigated. All distributed deadlock detection algorithms have a common goal — to detect cycles that span several sites in a distributed manner — yet they differ in the ways they achieve this goal. The following are the most salient characteristics of these algorithms:

• *The form in which the algorithm maintains and passes around information about the process-resource interaction.* Goldman's algorithm uses *lists*, Hass-Mohan's and Obermarck's use *strings*, Isloor-Marsland's uses *sets*, and Menasce-Muntz's uses the *condensed TWF graph.*

• *The way the algorithm conducts the search for cycles.* Obermarck's algorithm sends lists of the edges of a directed path in the state graph; Chandy et al.'s and Sinha-Natarajan's circulate a probe message along the edges of the state graph; Menasce-Muntz's passes the condensed TWF graph; Mitchell-Merritt's passes a label.

• *The amount of information available about a deadlock when it is detected.* In the algorithms of Chandy et al., Menasce-Muntz, Sinha-Natarajan, and Sugihara et al., a process that detects a deadlock knows that it is deadlocked but does not know all the other processes involved in the deadlock; in Goldman's and Haas-Mohan's algorithms, a process that detects a deadlock knows all the other processes involved in that deadlock.

Although several deadlock detection algorithms have been proposed for distributed systems, a number of issues remain to be addressed. Future research should focus on efficient resolution of deadlocks, correctness of distributed deadlock detection, modeling and performance analysis of deadlock detection algorithms, and the probability of deadlocks in distributed systems. □

Acknowledgments

The author is deeply indebted to Virgil Gligor of the University of Maryland, T.V. Lakshman of Bellcore, and seven anonymous referees whose comments on an earlier version were instrumental in improving the quality of presentation and enhancing the technical content of this article. The author is also thankful to Bruce Shriver, editor-in-chief of *Computer*, for his encouragement and valuable suggestions for revising the article.

References

1. K.M. Chandy, J. Misra, and L.M. Haas, "Distributed Deadlock Detection," *ACM Trans. Computer Systems*, May 1983, pp. 144-156.
2. Edgar Knapp, "Deadlock Detection in Distributed Database Systems," *ACM Computing Surveys*, Dec. 1987, pp. 303-328.
3. D.E. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. Software Engineering*, May 1979, pp. 195-202.
4. V.D. Gligor and S.H. Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Trans. Software Engineering*, Sept. 1980, pp. 435-440.
5. G.S. Ho and C.V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Trans. Software Engineering*, Nov. 1982, pp. 554-557.
6. B. Goldman, "Deadlock Detection in Computer Networks," Tech. Report MIT/LCS/TR-185, MIT, Cambridge, Mass., Sept. 1977.
7. L.M. Haas and C. Mohan, "A Distributed Detection Algorithm for a Resource-Based System," Research Report, IBM Research Laboratory, San Jose, Calif., 1983.
8. R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Trans. Database Systems*, June 1982, pp. 187-210.
9. M.K. Sinha and N. Natarajan, "A Priority-Based Distributed Deadlock Detection Algorithm," *IEEE Trans. Software Engineering*, Jan. 1985, pp. 67-80.
10. S.S. Isloor and T.A. Marsland, "An Effective On-line Deadlock Detection Technique for Distributed Database Management Systems," *Proc. Compsac 78*, Nov. 1978, pp. 283-288.
11. D.P. Mitchell and M.J. Merritt, "A Distributed Algorithm for Deadlock Detection and Resolution," *Proc. ACM Conf. Principles of Distributed Computing*, Aug. 1984, pp. 282-284.
12. K. Sugihara et al., "A Distributed Algorithm for Deadlock Detection and Resolution," *Proc. Fourth Symp. Reliability in Distributed Software and Database Systems*, Oct. 1984, pp. 169-176.
13. A.L. Choudhary et al., "A Modified Priority-Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Trans. Software Engineering*, Jan. 1989, pp. 10-17.
14. D.J. Badal, "The Distributed Deadlock Detection Algorithm," *ACM Trans. Computer Systems*, Nov. 1986, pp. 320-337.
15. G. Bracha and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection," *Proc. ACM Symp. Principles of Distributed Computing*, Aug. 1984, pp. 285-301.
16. J. Gray et al., "A Straw-Man Analysis of the Probabilty of Waiting and Deadlocks in a Database System," IBM Research Report, 1981.

Additional readings

Theory

E.G. Coffman et al., "System Deadlocks," *ACM Computing Surveys*, June 1971, pp. 66-78.

R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, Dec. 1972, pp. 179-195.

O. Wolfson and M. Yannakakis, "Deadlock Freedom (and Safety) of Transactions in a Distributed Database," *Proc. Fourth ACM Sigact/Sigmod Symp. Principles of Database Systems*, 1985.

Algorithms

A.N. Chandra et al., "Communication Protocol for Deadlock Detection in Computer Networks," *IBM Technical Disclosure Bulletin*, Vol. 16, No. 10, Mar. 1974, pp. 3471-3481.

L.M. Haas, "Two Approaches to Deadlock Detection in Distributed Systems," PhD Dissertation, Dept. of Computer Science, Univ. of Texas at Austin, 1981.

W. Tasi and G. Belford, "Detecting Deadlocks in Distributed Systems," *Proc. IEEE Infocom*, 1982, pp. 89-95.

A.K. Elmagarmid, "Deadlock Detection and Resolution in Distributed Processing Systems," PhD Dissertation, Dept. of Computer and Information Science, Ohio State Univ., Columbus, Ohio, 1985.

A. N. Choudhary, "Two Distributed Deadlock Detection Algorithms and Their Performance," Master's Thesis, Dept. of Electrical and Computer Engineering, Univ. of Massachusetts, 1986.

N. Natarajan, "A Distributed Scheme for Detecting Communication Deadlocks," *IEEE Trans. Software Engineering*, Apr. 1986, pp. 531-537.

I. Cidon et al., "Local Distributed Deadlock Detection by Cycle Detection and Clustering," *IEEE Trans. Software Engineering*, Jan. 1987, pp. 3-14.

B. Awerbuch and S. Micali, "Dynamic Deadlock Resolution Protocols," *Proc. 27th Annual Symp. Foundations of Computer Science*, Oct. 1987, pp. 196-207.

M. Roesler and W.A. Burkhard, "Resolution of Deadlocks in Object-Oriented Distributed Systems," *IEEE Trans. Computers*, Aug. 1989, pp. 1212-1224.

B.A. Sanders and P.A. Heuberger, "Distributed Deadlock Detection and Resolution with Probes," to appear in *Proc. Third Int'l Workshop on Distributed Algorithms*, Sept. 26-28, 1989, Nice, France.

Survey

S.S. Isloor and T.A. Marsland, "The Deadlock Problem: An Overview," *Computer*, Sept. 1980, pp. 58-77.

M. Singhal, "Deadlock Detection in Distributed Systems: Status and Perspective," Tech. Report No. OSU-CISRC-TR-86-10, Dept. of Computer and Information Science, Ohio State Univ., Columbus, June 1986.

A.K. Elmagarmid, "A Survey of Distributed Deadlock Detection Algorithms," *ACM Sigmod Records*, Sept. 1986.

Correctness

J.R. Jagannathan and R. Vasudevan, "Comments on Protocols for Deadlock Detection in Distributed Database Systems: Corrigenda," *Trans. Software Engineering*, May 1983, p. 271.

G. Wuu and A. Bernstein, "False Deadlock Detection in Distributed Systems," *IEEE Trans. Software Engineering*, Aug. 1985, pp. 820-821.

A.K. Elmagarmid et al., "A Distributed Deadlock Detection and Resolution Algorithm and Its Correctness," *IEEE Trans. Software Engineering*, Oct. 1988, pp. 1443-1452.

K. Shafer and M. Singhal, "A Correct Priority-Based Probe Algorithm for Distributed Deadlock Detection and Resolution and Proof of Its Correctness," Tech. Report No. OSU-CISRC-4/89-TR16, Ohio State Univ., Dept. of Computer and Information Science, Apr. 1989.

Performance

J.R. Jagannathan and R. Vasudevan, "A Distributed Deadlock Detection and Resolution Scheme: Performance Study," *Proc. Third Int'l Conf. Distributed Computing Systems*, 1982, pp. 496-501.

Probability of Deadlocks

C.A. Ellis, "On the Probability of Deadlocks in Computer Systems," *Proc. Fourth Symp. Operating Systems Principles*, Oct. 1973, pp. 88-95.

A.W. Shum and P.G. Spirakis, "Performance Analysis of Concurrency Control Methods in Database Systems," *Performance 81*, 1981, pp. 1-19.

W. Massey, "A Probabilistic Analysis of a Database," *Performance Evaluation Review*, Vol. 14, No. 1, May 1986, pp. 141-146.

Mukesh Singhal is an assistant professor of computer and information science at Ohio State University, Columbus. His research and teaching interests are distributed systems, distributed database systems, and performance modeling of computer systems. From 1981 to 1985, he served as a research assistant and instructor in the Department of Computer Science, University of Maryland, College Park.

Singhal received a bachelor of engineering degree with distinction in electronics and communication engineering from the University of Roorkee, Roorkee, India, in 1980, and a PhD degree in computer science from the University of Maryland, College Park, in May 1986. He is a member of the Computer Society and the IEEE.

Singhal's address is Ohio State University, Dept. of Computer and Information Science, 2036 Neil Avenue Mall, Columbus, OH 43210.