ON ROBUSTNESS OF DETECTION ALGORITHMS FOR DISTRIBUTED ETC+++
ON ROBUSTNESS OF DEADLOCK DETECTION ALGORITHMS FOR DISTRIBUTED COMPUTING SYSTEMS

Dushan Z. Badal and Michael T. Gehl

February 1982

Approved for public release; distribution unlimited

Prepared for:
Naval Postgraduate School
Monterey, California 93940
The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:

DUSHAN Z. BADAL
Assistant Professor of Computer Science

MICHAEL T. GEHL, LCDR, USN

Reviewed by: Released by:

GORDON H. BRADLEY, Chairman
Department of Computer Science

WILLIAM M. TOLLES
Dean of Research
### On Robustness of Deadlock Detection Algorithms for Distributed Computing Systems

**Authors:** Dushan Z. Badal and LCDR Michael T. Gehl, USN

**Abstract:**
In this paper, we investigate the robustness of several deadlock detection algorithms for distributed computing systems. We analyze the behavior of each algorithm in the presence of two classes of failures—lost messages and single site failures. In the case of single site failure, we consider six different types of sites depending on how they can participate in deadlock and deadlock detection. The observation and conclusions made in this paper have been submitted for publication and if accepted, will be copyrighted for publication.

**Key Words:** Robustness, Reliability, Deadlock Detection, Distributed Computing Systems, Failure.
paper are intended to show how robust the present algorithms are and to provide an insight and better understanding of distributed algorithms robustness.
In this paper we investigate the robustness of several deadlock detection algorithms for distributed computing systems. We analyze the behavior of each algorithm in the presence of two classes of failures - lost messages and single site failures. In the case of single site failure we consider six different types of sites depending on how they can participate in deadlock and deadlock detection. The observation and conclusions made in this paper are intended to show how robust the present algorithms are and to provide an insight and better understanding of distributed algorithms robustness.
1. INTRODUCTION.

There have been many algorithms published for deadlock detection, prevention or avoidance in centralized multiprogramming systems. The problem of deadlock in those systems has been essentially solved. In the past decade there has been considerable work done on distributed computer networks and multiprocessor systems. Both of these are predecessors of distributed computing systems which are presently a focus of intensive research and development in academia and industry. Many techniques for concurrency control, reliability/recovery or security developed for centralized (or single CPU) systems have been or are being adopted and adapted for distributed computing systems. For example, there is a tendency to use locking as a general synchronization technique in distributed systems and its special variant, two-phase locking, for distributed database systems. Up until recently it has been argued that the frequency of deadlock occurrence in existing applications is so low that the problem of deadlock in distributed systems is not very important and therefore can be managed by adopting techniques developed for centralized systems. However, it has become recently apparent that deadlocks may be a problem in the future as we see new applications featuring large processes and/or many concurrent processes or transactions[GRSEG]. As an example of such new applications we mention information utility systems which service concurrently hundreds or perhaps thousands of TV users.

The distributed computing systems are characterized by the absence of global memory and by message transmission delays which are not negligible. Additionally, the processes operating at the same or different sites can communicate with each other, and can share resources. If locking is used as the synchronization technique, then
the last two items raise the problems of deadlock occurrence in distributed systems, and the first two characteristics of distributed systems make it much more difficult to detect, avoid or prevent than in the earlier multiprogramming centralized computing systems.

Deadlock prevention and avoidance algorithms for a distributed computing systems are not efficient. Prevention can be accomplished by not allowing concurrent processing, by assigning priorities and allowing preemption, by requiring a process to acquire all resources it will need before it starts, or by having no locks. Requiring sequential execution in a distributed system is a gross waste of resources. Having prioritized processes will result in lower-prioritized processes being restarted many times, with a major degradation in system efficiency. Dynamic prioritization would be a complex algorithm by itself. A process may be unable to determine its minimum set of resources, and therefore would have to acquire the set of all probable and possible resources, even though it may not need them. In addition, in systems in which messages are treated as resources, it is impossible to determine in advance which messages will be required. Having no locks may result in database inconsistencies, assuming a non-optimistic concurrency controller. Similarly, deadlock avoidance algorithms, which either calculate a 'safe path' [3,4,7] or never wait for a lock [5,6,8,9] are also inefficient. Safe path algorithms require a non-trivial execution time, and must be done each time a resource request is to be granted. Never waiting for a lock is inefficient when deadlock is a rare occurrence. Thus, in distributed computing systems, deadlock detection and resolution algorithms must be used.

There are four criteria that any deadlock detection algorithm for distributed computing systems must meet. They are: 'correctness, 2'
robustness, 3) performance, and 4) practicality. Correctness refers to the ability of the algorithm to detect all deadlocks, and the ability to not detect any false deadlocks. Robustness refers to the ability of the algorithm to be correct even in the presence of anticipated faults. This includes the ability to detect deadlocks even when a site fails or loses communications while the deadlock detection algorithm is being executed. The performance of the algorithm refers to its overhead - the delays between deadlock and detection, CPU time used, number of messages required, etc. Practicality is closely related to performance. It refers to aspects such as complexity and cost.

Several different approaches are being used in current deadlock detection and resolution algorithms for distributed systems. Two major ones are centralized and distributed deadlock detection algorithms. Within the distributed class are two subclasses: 1) all or several sites execute the deadlock detection algorithm, and 2) only one site is actually executing, although the algorithm is resident in all sites and thus any site could execute the algorithm. It might be easier to view the algorithms as a continuum: fully centralized[GRA78], hierarchical[HEN79], distributed with a single site at a time executing the algorithm[JCL77], distributed with all sites involved in a possible deadlock executing the algorithm concurrently[HEN79], and distributed with all sites executing the algorithm concurrently[ISE78].

In this paper we investigate the robustness of several published deadlock detection and resolution algorithms for distributed systems. The motivation for our work comes from three facts. First, very few authors investigated robustness or reliability of deadlock detection
algorithms. Second, reliable deadlock detection and resolution for upcoming new distributed systems and applications is in our opinion an urgent, very important and as yet not satisfactorily resolved problem. Third, if there can be more than one deadlock being detected by the deadlock detection algorithm then it is reasonable to expect such algorithm to be robust, i.e., to continue executing and detecting all deadlocks even in the presence of failures which might have in effect broken one of the deadlocks being detected.

The paper is organized as follows. In section two, we discuss robustness of distributed systems. In section three, we analyze the robustness of several existing deadlock detection algorithms with respect to some single failures. In section four, we present our conclusions based on the analysis of section 3.

II. SOME THOUGHTS ON ROBUSTNESS IN DISTRIBUTED SYSTEMS.

In this paper we want to investigate the robustness of deadlock detection algorithms (DDA), i.e., we want to find out the impact of some single failures on such algorithms. In general, the DDA is invoked by two events—either whenever a process waits for a resource, or after a certain period of time has elapsed since the last DDA invocation. In the first case, deadlock is checked for whenever its possibility appears, and in the second case it is checked for periodically, i.e., regardless of whether its possibility exists.

The DDA can reside in one, several or all sites of the distributed computing system. When a triggering event for DDA occurs, then depending on a particular algorithm one, several or all sites will receive information from several or all sites. Such information consists of “who waits for whom and where”, and it can be represented in
arcs of the wait-for graph, strings, or lists of processes or transac-
tions. Upon receipt of such information one, several or all sites
attempt to reconstruct a global state of the distributed system, i.e.,
to generate a true snapshot of all or of all waiting processes in the
system.

The generation of such a true snapshot in the distributed system
is difficult because of lack of global memory and the message delays
which are not negligible and can vary considerably. The generation of
such a true snapshot, usually referred to as a global wait-for graph,
becomes even more difficult when we consider a possibility of failures
in the distributed system. Some system mechanisms have been designed
to be robust or reliable. For example, some concurrency control or
synchronization mechanisms for distributed databases and transaction
processing systems are based on two phase locking, which has been made
robust by incorporating atomicity by using two phase commit protocols.
The two phase commit protocol supports not only the atomicity of trans-
sactions but also it supports the robustness of locking, i.e., the
robustness of concurrency control mechanisms. In particular what
makes the concurrency control which uses locking robust is the need to
lock and unlock resources in a robust way, i.e., either all
lock/unlock operations for a given process or transaction occur or
none occur. Thus in some sense, the robustness of concurrency control
is meant to support the atomicity of placing and releasing a set of
locks needed by a process. In other words, the robustness of concen-
tracy control means that no dangling locks or locked resources are
left behind the terminated or committed process, even in the presence
of some failures. It is interesting to note that although deadlock
detection is a part of concurrency control based on locking, there has
been no attempt to provide for or even to investigate the robustness of deadlock detection mechanisms. The most likely explanation for this is that from the concurrency control point of view, the inability of the process to lock a needed resource is an exception to be handled by another mechanism, i.e., a deadlock detection algorithm 'DDA'.

The proper way to see the DDA is as another transaction running under the concurrency control mechanism, as it reads and shares lock tables with concurrency controllers and other transactions. Moreover, DDA is a special transaction which operates on special data it creates solely for deadlock detection, e.g., wait-for graphs. Such data, we'll call it deadlock data, is internal to each invocation of DDA transaction and is erased after its execution. Moreover, such deadlock data is not shared by any other DDA transaction invocations and therefore they need not be locked. This means that the robustness required of DDA transactions is of a somewhat different kind than the robustness of transactions operating on shared database data. Thus it makes sense that the DDA transaction does not need to use two phase commit to assure its robustness. The question then is what kind of robustness or fault-tolerance we need for DDA transactions and this is precisely the problem we are addressing in this paper.

We consider the following informal model of DDA transaction execution. The DDA is invoked by a concurrency controller at a site at which a database transaction can not acquire locks which are being held by another transaction(s). The DDA transaction executes at one, several or all sites depending on the DDA itself and the deadlock topology. During its execution the DDA transaction should exhibit the extensibility property, i.e., it either executes correctly or it does not execute at all. The results of DDA transaction execution are not
messages to the concurrency controller which has triggered it:

1. Proceed - because of
   a) no deadlock
   b) deadlock detected but another transaction was selected as a victim for back-up

2. Abort - because of
   a) deadlock detected and you are the victim.
   b) DDA transaction failed, i.e., it did not execute.

The situation we investigate in this paper is when DDA transactions fail or should not fail, i.e., how robust the existing DDA's are or should be. In this paper we consider only two classes of single failures. First, we investigate the impact of lost messages and second, we investigate the impact of one site failures, or identically one site partitions on DDA behavior. We investigate the impact of lost messages because not all distributed systems may support reliable delivery of messages, several algorithms treat messages as resources[13,17], and in some applications, acknowledgements cannot be sent.

III. RELIABILITY ANALYSIS OF DEADLOCK DETECTION ALGORITHMS

In this section, we examine four published deadlock detection algorithms for distributed computing systems with respect to the presence of the two classes of failures (lost messages and site failures) discussed in section two. Although very few of them have already been shown to be correct when no failures or errors occur, we feel that their robustness is nevertheless worth analyzing. The assumptions made by each author will be discussed in the context of how robust the algorithm is. We will analyze each DDA by executing it in the following environment.

There are four sites in the system, each of which has a single
resource and a single transaction. (These restrictions merely make the example simpler, they are not required for the analysis.) The initial system status is shown in Figure 1. Transaction $T_1$ at site A holds resources $R_2$ and $R_3$ and is waiting for resource $R_4$. Transactions $T_2$ and $T_3$ hold no resources. Transaction $T_4$ at site C holds resource $R_4$, but is active. We assume that the deadlock detection activity resulting from $T_1$ waiting for $R_4$ has been completed, so there is currently no deadlock detection activity in the system. For the algorithms which require global timestamps, we assign timestamp 'TS' $t_1$ to the $T_1< R_2$ assignment, $t_2$ to the $T_4< R_4$ assignment, $t_3$ to the $T_4< R_3$ assignment, and $t_4$ to the $T_1-> R_4$ request. Now at some time $t_6$, transaction $T_4$ requests $R_3$, resulting in a global deadlock $T_1-> T_4-> T_1$.

![Figure 1](image)

In the case of a site failure, we distinguish the following cases. a) A site can have a transaction involved in a deadlock but not be involved in deadlock detection, b) a site can have a transaction involved in a deadlock and be involved in detection, c) a site can have a resource involved in a deadlock and not be involved in detection, d) a site can have a resource involved in a deadlock and be involved in detection, e) a site can be involved in deadlock detection but in no way involved in a deadlock. Not all of these possibilities exist with each algorithm.
A. THE DISTRIBUTED DEADLOCK DETECTION ALGORITHM OF GOLDMAN.

In [GOLD], Goldman presents two deadlock detection algorithms. Only the distributed version will be considered in this paper. A Process Management Module (PMM) at each site handles resource allocation and deadlock detection. An 'ordered blocked process list' (OBPL) is a list of process names, each of which is waiting for access to a resource assigned to the preceding process in the list. The last process in the list is either waiting for access to the resource named, or it has access to that resource. An OBPL is created each time a PMM wants to see if a blocked process is involved in a deadlock. In the distributed algorithm, an OBPL is passed from a PMM to another PMM which has information either about a resource or a transaction in the OBPL which is needed to expand the OBPL. Each PMM adds the information it knows, and either detects a deadlock, detects a non-deadlocked state, or passes the OBPL to another PMM for further expansion. The terms process and transaction will be used synonymously in the analysis of this DDA. If several transactions are waiting on one transaction, multiple copies may be made of the OBPL and sent to each site having one of those waiting transactions. Processes can be in either of 2 states, active or blocked (waiting). A blocked process could be waiting for a database object, message text from another process or message text from an operator. A process is active if it is not blocked. In the algorithm, RX and RX are temporary variables representing a process or resource. The steps of the algorithm are:

1. Set RX to the value contained in the resource identification portion of the OBPL. If RX represents a local resource, go to 2. Otherwise, go to 3.

2. Verify that the last process aided to OBPL is still waiting for RX. If so, go to 3; otherwise, halt.
3. Let RX be process controlling RX. If RX is already in CBPL, then there is a deadlock. If not, go to 1.

4. If RX is local to current PM, go to 5, otherwise go to 7.

5. If RX is active, there is no deadlock. Discard CBPL and halt. Otherwise go to 6.

6. Add RX to CBPL and go to 'O.

7. Add RX and RX to CBPL. Send CBPL to PM of site in which RX resides. Halt.

8. Verify that last process in CBPL still has access to RX. If not, there is no deadlock, so discard CBPL and halt. If so, go to 9.

9. If last process in CBPL is active, there is no deadlock, so discard CBPL and halt. Otherwise go to 'O.

'O. Call resource for which last process is waiting RX. If RX is local, go to 3. Otherwise go to '1.

'I. Place RX in CBPL and send CBPL to PM of site in which RX resides. Halt.

Figure 2 shows the actions taken at each site during the execution of the DDA following the request by P4 for resource R3. The numbers refer to the current step being executed by the DDA. As can be seen, the algorithm correctly detected the resulting deadlock, in an environment of no faults. If, however, a message is lost (in our example, either the CBPL sent from site C to A, or the CBPL sent from A to C), the necessary information to detect the deadlock will be lost, and the algorithm will fail to detect an existing deadlock.
Goldman's algorithm allows the following types of sites discussed previously: type b (a site can have a transaction involved in deadlock and the site is involved in detection), type i (a site can have a resource held by a transaction involved in deadlock and the site will be involved in deadlock detection), and type c (a site can have a resource held by a transaction involved in a deadlock and not be involved in deadlock detection). A site could also be in several of the categories above, depending on the complexity of the system state. For example, site D could be considered a type b or type i site. If a site of type b (sites A or C in our example) fails during execution of the CDA, the behavior could be different depending on the time of the failure. If the failure occurred at site A before site C sent the CEBPL to site A, site C would realize that site A had failed. The algorithm includes no procedure for this occurrence, so the behavior would be dependent on the underlying system. If the failure at site A occurred after it received the CEBPL, all deadlock detection
activity will cease, because only site A was currently involved in deadlock detection. A system timeout mechanism would eventually abort the transactions involved in the deadlock. A failure at site I would have the same effect as at site A.

If a site of type i (site C in our example) failed, the time of the failure would again determine the behavior of the DDA. If the failure occurred before site I sent the CBPL to site A, deadlock detection activity would cease without deadlock having been detected. If the CBPL had been sent, however, deadlock detection would continue at sites A and D (sequentially) with site D detecting a deadlock. The failure of site C would not have been critical after the CBPL had been sent. The effect of a type c site (site C in our example) failing would have no effect on the behavior of the DDA, because the fact that R2 is held by T1 is not used or known by the DDA at any site.

There are essentially two types of CBPL's created by this DDA. The first type, call it W, is when a process is waiting, but is not involved in a deadlock. This CBPL is subsequently discarded. The second type, call it D, is one which will eventually show a deadlock cycle. If there are n transactions involved in a deadlock cycle, this DDA will create from 1 to n type D CBPL's. In our example, only one was created. If the request by T1 for resource R4 happened simultaneously with the request by T4 for resource R7, two CBPL's would have been created which would have resulted in two sites independently detecting the same deadlock, vice the one site in our example. Thus the robustness of this algorithm with respect to a single site failure is related to the ratio of the number of D type CBPL's created to the number of transactions involved in the deadlock. This ratio is however determined by the sequencing or timing of transactions messages
blocked resources. Such sequencing is of random nature. A ratio of 1 would provide the highest degree of robustness. When only a single CEPZ is created, the robustness of the DDA is very similar to that of a centralized DDA; a single site failure can stop deadlock detection activity. We conclude that the robustness of this DDA can be analyzed but it can not be predicted.

B. THE MENASCE-Hunts DISTRIBUTED ALGORITHM

In [MEN79], Menasce and Hunts presented a distributed deadlock detection algorithm. Jilior and Shattuck [JIL80] presented a counter example which showed the algorithm to be incorrect in that it failed in some cases to detect a deadlock. They also proposed a modification to the algorithm which they thought would make it correct, but they felt the algorithm was impractical. In [RSA82], Csaí and Seldford show that the algorithm as modified by Jilior and Shattuck is also incorrect. Nevertheless, we will investigate the enhanced algorithm (i.e., its modified version as suggested by Jilior and Shattuck) in the presence of errors.

The algorithm constructs a Transaction-Waits-For (TWF) graph at originating sites of transactions which are potentially involved in the deadlock being detected, and at sites at which some transaction could not acquire a resource. Nodes in the TWF graphs represent transactions. An edge (ti, tj) indicates that transaction ti is waiting for transaction tj. A non-blocked transaction is a transaction that is not waiting and is represented in the TWF graph by a node with no outgoing arcs. A blocked transaction is waiting for some transaction to finish. A 'Blocking set' is defined as the set of all non-blocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction tij.
A pair \( (T, T') \) is a 'blocking pair' of \( T \) if \( T' \) is in the blocking set of \( T \). A 'Potential Blocking set' consists of all waiting transactions that can be reached from \( T \) [HILSC]. \( \text{Sorig}(T) \) means the site of origin of transaction \( T \). \( S_0 \) is the site currently executing the algorithm.

The rules which define the enhanced algorithm, as executed at site \( S_0 \), are:

**Rule 1:** When a transaction \( T \) requests a nonlocal resource it is marked 'waiting'.

**Rule 1:** The resource \( R \) at site \( S_0 \) cannot be allocated to transaction \( T \) because it is held by \( S_1, \ldots, S_k \). Add an arc from \( T \) to each of the transactions \( T', \ldots, T_k \). If there is then a cycle formed in the TWF graph, deadlock has been detected. Otherwise, for each transaction \( T' \) in blocking set \( T \), send the blocking pair \( (T, T') \) to \( \text{Sorig}(T') \) if \( \text{Sorig}(T') \neq S_0 \) and to \( \text{Sorig}(T) \) if \( \text{Sorig}(T') = S_0 \). Form a list of potential blocking pairs associated with \( T \).

**Rule 2:** A blocking pair \( (T, T') \) is received. Add an arc from \( T \) to \( T' \) in the TWF graph. If a cycle is formed, then a deadlock exists.

**Rule 2:** If \( T \) is blocked and \( \text{Sorig}(T') = S_0 \), then for each transaction \( T'' \) in the blocking set \( T' \), send the blocking pair \( (T'', T') \) to \( \text{Sorig}(T') \) if \( \text{Sorig}(T'') = S_0 \).

**Rule 2:** If \( T \) is waiting and \( \text{Sorig}(T') = S_0 \), then for each potential blocking pair \( (T'', T') \) send the blocking pair \( (T'', T') \) to \( \text{Sorig}(T') \) if \( \text{Sorig}(T'') = S_0 \). Then, discard the potential blocking pairs \( (T'', T') \) and erase the 'waiting' mark of \( T \).

Figure 3 shows the actions taken at each site during the execution of the CDA following the request by \( T_4 \) for resource \( R_3 \). As can be seen, the deadlock was correctly detected by site \( A \), in absence of failures. If the request message \( R_4, R_3 \) from site \( D \) to site \( C \) was lost, however, deadlock detection activity would cease. If the blocking pair \( (T_4, T_1) \) from site \( D \) to site \( B \) was lost, site \( A \) would still detect the deadlock. If, however, the blocking pair \( (T_1, T) \) from site \( D \) to site \( A \) was lost, site \( D \) would apply rule 2. Neither rule 2.1 or 2.2 applies, so deadlock detection activity would cease.
Figure 3.

This algorithm allows sites of types b, c, d and e, although our example does not include a site of type e. If a type b site (one having a transaction involved in the deadlock and the site is also involved in detection) failed, in our example site A (or site D), the behavior of the algorithm is dependent on the time of failure. If site A failed before receiving the blocking pair (T4, T1), site C would recognize the failure, but its action is not specified in the rules of the DDA. Site D would not detect the deadlock for the same reason as if the message from site C to site A was lost. If, however, the failure occurred after site A received the blocking pair, deadlock detection activity would continue (at site D) but deadlock would not be detected. A failure of site D, also a type b site, at any time, would have no effect on detecting the deadlock in this example. If a type c site failed (site E), it would have no effect on detecting the deadlock. If a type i site (site I) failed, the time of its failure would determine the behavior of the DDA. If it failed before sending...
the blocking pair to sites A and C, deadlock detection activity would cease. If it failed after sending those messages, it would have no effect on detecting the deadlock.

For our example, this algorithm behaved surprisingly similarly to Goldman's algorithm in almost all types and timings of failures. This may just be an anomaly found in small deadlock cycles, because in longer and more complex scenarios, it would appear that more sites would be involved in detection, and that there would be some duplication of information. As the number of transactions (and resources) involved in a deadlock cycle increases, more blocking pairs and potential blocking pairs will be sent to more sites, i.e., the number of sites detecting the deadlock is increasing with the number of transactions involved in the deadlock and with the deadlock topology (or complexity). Thus there will be more chance of a deadlock being detected, as more parallel detection activity will be in progress. It appears, then, that as the site and complexity of deadlock increases, the robustness of this algorithm increases. However, as pointed out by Iligor and Shattuck, the effect which Iligor and Shattuck point out of rule 2.2 discarding information too early may have some impact on the increased robustness.

3. OBERMARCK'S DISTRIBUTED DEADLOCK DETECTION ALGORITHM.

In [CEEO], Obermarck presents a distributed deadlock detection algorithm. A centralized algorithm is presented by Obermarck and Seeri in [CEEO], but it is not discussed here because no mention is made in that paper about a backup capability if the site containing the centralized deadlock detector fails. Obermarck's distributed algorithm constructs a transaction-waits-for (TWF) graph at each site. Each site conducts deadlock detection simultaneously, passing
information to one other site. Deadlock detection activity at a site may become temporarily inactive until receipt of new information from another site. Chermerock states that in actual practice, synchronization (not necessarily precise) between sites would be roughly controlled by an agreed-upon interval between deadlock detection iterations, and by timestamps on transmitted messages. Nodes in the graph represent transactions, and edges represent a transaction-wait-for-transaction (TWF) situation. A 'String' is a list of TWF information which is sent from one site to one or more sites. A transaction may migrate from site to site, in which case an 'agent' represents the transaction at the new site(s). A communication link is also established between agents of a transaction. These communication links are represented by a node called 'External.' An agent which is expected to send a message is shown in the TWF graph by EX->C, while an agent waiting to receive is shown by C->EX. Although Chermerock's algorithm includes the resolution of deadlocks, only the detection part will be considered in this paper. Transaction ID's are network unique names for transactions, and are lexically ordered. (For example, T1 < T2 < T3). The steps performed at each site are:

1. Build a TWF graph using transaction to transaction wait-for relationships.

2. Obtain and add to the existing TWF graph any 'strings' transmitted from other sites.
   a. For each transaction identified in a string, create a node in the TWF if none exists in this site.
   b. For each transaction in the string, starting with the first (which is always 'external'), create an edge to the node representing the next transaction in the string.

3. Create wait-for edges from 'external' to each node representing a transaction's agent which is expected to send on a communication link.

4. Create a TF edge from each node representing a transaction's
agent which is waiting to receive from a communication link to 'external.'

5. Analyze the graph for cycles.

6. After resolving all cycles not involving 'external', if the transaction ID of the node for which 'external' waits is greater than the Transaction ID of the node waiting for 'external', then

a. Transform the cycle into a string which starts with 'external', followed by each transaction ID in the cycle, ending with the transaction ID of the node waiting for 'external'.

b. Send the string to each site for which the transaction terminating the string is waiting to receive.

In his proof of correctness, Obermarck shows how the algorithm can detect false deadlocks because a string received at a site may no longer be valid when it is used. He discusses two methods of handling false deadlocks; treat them as actual deadlocks (if they don't occur too often), or verify them by sending them around the network and have each site verify them.

Figure 4

Figure 4 shows a global picture of the system, including the communication links established between agents, for the initial conditions of our example. The agents of T1 at sites A and B have
performed work (used R2 and R3), and are waiting for the next request from T1 at site A. T1 at site A is waiting for its agent at site D, which is in resource-wait for T4. Figure 5 shows the actions of this algorithm in an environment of no errors. As can be seen, it successfully detects the deadlock.

```
Site A  Site B  Site C  Site D

T4 requests R3

An agent of T4 is formed

1, 3, 4: each site starts deadlock detection and builds WF graph.

T1 -> EX  T1 -> EX  T1 -> EX -> T4  EX -> T1 -> T4

5: list elementary cycles

T1 -> EX -> T1  EX -> T4 -> T1 -> EX  EX -> T1 -> T4 -> EX

6: form string

(EX, T4, T1)

Send to A.

2: T4 <- T4

T1 -> EX

5: Form string

(EX, T4, T1)

Send to D.

2: T4 <- T4

5: Deadlock detected
```

Berzwarek assumes that messages sent are received. This is essential to the correctness of this DDA, because it is easy to see what happens if a message is lost. If the string (EX, T4, T1) from site C to A, or from A to D were lost, deadlock detection activity would...
caze; without detecting a readlock. The use of agents to represent
transactions has been
3.3, a site with-
Atohe
"nodes of
type a or b,
if we substitute agents for transactions in our definitions at the
beginning of this section. It will be seen that the
number of transactions can be decreased, which would
cause deadlock detection activity to cease, depending on the time
of the failure. For example, if site 2 failed before the string
was sent to site A, deadlock detection activity would cease. If site
A (or D) failed before the string
was sent to them, the transmitting site would recognize the
failure, but its action in that eventuality is not included in
the steps of the DDA. If site 2 failed after sending the
string, the detection activity would continue, and the deadlock
would be detected.
This DDA appears to be not significantly more robust than the
earlier two. All sites contain and retain more information in
their and all sites start detection activity simultaneously,
and neither stay involved for the entire detection process. The use
of the lexical orders of nodes was for optimization of the number
of necessary restore times at the time of test. This would have
followed sites A and D to somehow detect deadlock. The DDA
would be clearer, more robust, but the overhead would be that of
its existing form, the DDA's robustness is similar to that of
essential situation.
D. THE ALGORITHM OF TSAI AND BELFORD.

In [TSA82], Tsai and Belford present a distributed deadlock detection algorithm. They utilize a "Reduced Transaction-Resource" (RTR) graph, which contains only a subset of the transaction resource graph, but has all relevant TWF edges. Nodes in the RTR graph can be transactions or resources. The algorithm uses a concept the authors call a "reaching pair", which is the basic unit of information passed from site to site. If a path \( T_1 T_2 \ldots T_n \) can be formed by following TWF edges, and if there is a request edge \((T_n, R_m)\), then \( T_i \) "reaches" \( R_m \), and \((T_i, R_m)\) is a "reaching pair." Five types of messages are sent between sites: reaching messages, nonlocal request messages, allocation messages, release-request messages, and releasing messages. The non-local request messages include a list of all resources currently held by the requesting transaction. Five different types of edges are distinguished in the RTR graph: requesting edges, allocation edges, TWF edges, resource reaching edges and transaction reaching edges. A global timestamp is also used to establish an ordering of events. This timestamp is used on allocation, request and reaching messages, and on allocation and reaching edges in the RTR graph. The notation used in the algorithm is:

- \( TS(X) \): timestamp of a message
- \( TS(C) \): current system time
- \( TS(A) \): timestamp of an allocation edge
- \( TS(R) \): timestamp of a reaching edge
- \( /=/ \): not equal to
- \( S_{orig} \): Site of origin

The steps of the algorithm (as executed at site \( 2k \)) are:

**Step 1:** A transaction \( T \) enters the system requesting a nonlocal resource \( R \). Add request edge \((T, R)\) to RTR graph. Send request message \((T, R, S_{orig})\) to \( S_{orig} \), where \( S_{orig} \) is the set of all resources allocated to \( T \), and \( TS(R) = TS(T) \). \( S_{orig} \) has each \( TS(A) \) attached, and \( S_{orig} \) is empty if \( T \) holds no resources.
Step 1a: A transaction $T$ releases a nonlocal resource $R$. Erase edge $(R, T)$ in the RTR graph. Send a release-request message $(R, T)$ to $S_{orig}(T)$.

Step 2: A transaction $T$ enters system requesting local resource $R$. Go to step 4.

Step 2a: A transaction $T$ releases a local resource $R$. Erase edge $(R, T)$ in RTR graph. If there is any transaction $T'$ waiting for $R$, then begin

  Add allocation edge $(R, T')$ to RTR graph with $TS(A) = TS(C)$. Send allocation message $(R, T', TS)$ with $TS(A) = TS(C)$ to $S_{orig}(T')$ if $S_{orig}(T') 
eq Sk$. end.

Step 3: A request message $(T, R', R, TS)$ is received. Add allocation edges $(R, T)$ for each Ri in $R'$ to RTR graph. Go to step 1.

Step 3a: A release-request message $(R, T)$ is received. Erase allocation edge $(R, T)$ in RTR graph. Send releasing message $(R, T)$ to $S_{orig}(T)$. If there is any transaction $T'$ waiting for $R$, then begin

  Add allocation edge $(R, T')$ to RTR graph with $TS(A) = TS(C)$. Send allocation message $(R, T', TS)$ to $S_{orig}(T')$ if $S_{orig}(T') 
eq Sk$. end.

Step 4: If $R$ is not held by any transaction, then begin

  Add allocation edge $(R, T)$ with $TS(A) = TS(C)$ to RTR graph. If $S_{orig}(T) 
eq Sk$, then send an allocation message $(R, T, TS)$ with $TS(A) = TS(C)$ to $S_{orig}(T)$. end.

else begin

  Add requesting edge $(T, R)$ to RTR graph. Suppose $R$ is held by transaction $T'$. Add edge $(T, T')$ to RTR graph. If there is a cycle, deadlock has been detected, else go to step 5. end.

Step 5: (reaching message generation step) If there are two edges $(T, R)$ and $(T', R')$ added to the graph, and if $T' = ... = T''$ is any path obtained by following the TWF and transaction reaching edges, then set $X = R''$ if $T''$ has outgoing edge to $R''$, else set $X = R$. For all transaction $T'$ in RTR graph reaching $X$ via $T$, do begin

  If $T'$ holds any resource $R'$ with $S_{orig}(T) 
eq Sk$, and $S_{orig}(R') 
eq Sk$, then send a reaching message $(T', X, TS)$ to $S_{orig}(R')$. If $S_{orig}(T) 
eq Sk$ and $T' = T$, then send a reaching message $(T', X, TS)$ to $S_{orig}(T)$. If $S_{orig}(T) 
eq Sk$ and $T = T'$ and $X = R''$ then send a reaching message $(T, X, TS)$ to $S_{orig}(T)$.

The TS in the reaching message is set to $TS(C)$ if triggered by a local request, and set to $TS(A)$ if the nonlocal request or reaching message otherwise.

Step 6: (An allocation message $(R, T, TS)$ is received) If $R$ is an entry in the graph, then begin

  Erase allocation edge $(R, T)$ and all reaching edges $(T, R)$ with $TS(A) < TS(C)$ and the corresponding edge $(T, T')$ and transaction reaching edges $(T', R')$. If
they exist, where $T' \neq T$. Change requesting edge $(T, R)$ to allocation edge $(R, T')$ with $TS(A) = TS(M)$ if $(T, R)$ exists, and for each resource reaching edge $(T''', R)$, add the transaction reaching edge $(T'', T)$. If $\text{Sorig}(T) = S_k$, wake up transaction $T$. end.

Step 6a: [A releasing message $(R, T)$ is received] If $\text{Sorig}(T) = S_k$, wake up transaction $T$. end.

Step 7: [A reaching message $(T, R, TS)$ is received] If there exists an allocation edge $(R, T')$ in the graph with $TS(M) < TS(A)$ and $T' \neq T$, then skip this step, else begin.
Add resource reaching edge $(T, R)$ to the RTR graph. If $R$ is held by transaction $T'$, then add the transaction reaching edge $(T, T')$ to the graph. If there is a cycle in the graph, there is deadlock (go to step 8), otherwise go to step 5. end.

Step 8: [a deadlock has been detected] Take appropriate action.

Figure 6 shows the starting WF graphs and the actions of the DDA resulting from the request by transaction $T_4$ for resource $R_3$. An important item to note is that as soon the request is made, step 1 adds sufficient information to the WF graph to detect a deadlock, but does not check for deadlock, so the request is sent to site $C$ and the algorithm continues. The obvious thing to do would be to add a check for a deadlock cycle in step one, but on closer analysis, this check may lead to detection of false deadlocks (if, for example, $T'$ had just released $R_3$ but the message had not yet been received by site $C$).
Therefore the algorithm in its present form will be analyzed. The only message sent by this algorithm in this example is the request message $(T_4, R_3, T_3, TS)$. If it was lost, the current algorithm would cease detection activity without detecting deadlock. In this instance, if the algorithm checked for deadlock in step 1, it would have been detected with no messages required.
For this DDA, sites can be of type b, i or e. Sites A and D are type b and sites B and C are type i. This example has no type e sites, but step 5 of the algorithm could send reaching messages to sites not involved at all. Those sites would execute a step or two of the algorithm, but not be intimately involved in the actual deadlock detection. In this example, a failure of sites A or B (types b and i respectively) would have no effect on the detection of the deadlock. The effect of a failure of site C before the reaching message was sent to it cannot be determined because the DDA includes no instructions for that event. A failure of site C after receiving the reaching message would result in a cessation of detection activity. If the algorithm were modified to include a role check in step 1, a failure of
site C at any time would have no effect on deadlock detection. The timing of the failure would also determine the behavior of the DDA if site D failed. If site D failed before sending the request message, detection activity would cease, while if the message had been sent, deadlock would still be detected.

For our example, this DDA appears to be about the same level of robustness as the other algorithms, except that each site contains and retains more information than in other DDA's. This indicates that it should be more robust. The algorithm in the case of our example was able to detect the deadlock with only the resource request message. As deadlock cycles become more complex, it appears that this algorithm will also become more robust, even more so than Obermarck's, because this DDA retains more information, and it will send reaching messages to any site potentially involved in the deadlock. Detection activity will occur simultaneously in those sites receiving reaching messages. The impact of the inclusion of a cycle detection in step 1 may have adverse effects on the correctness, but it might greatly enhance the robustness of the DDA.

IV. CONCLUSIONS

The algorithms discussed in the previous section can be loosely ranked by their robustness. Goldman's algorithm is the least robust, because it is always executed sequentially (unless the requests occur simultaneously, as discussed previously). Thus it is always dependent on a single node. Obermarck's algorithm starts deadlock detection simultaneously at all sites, and subsequently passes information in a lexical manner because of the message optimization. For our example, this resulted in a sequential detection, although for larger deadlock
cycles, it should have some parallel detection activity occurring. The Menasce-Huntz algorithm starts detection at the site where the deadlock occurred, and deadlock detection is subsequently conducted at sites which are potentially involved. The Tsai-Belford algorithm is invoked each time a resource is requested. Deadlock detection can appear concurrently at all sites potentially involved in the cycle. It appears more robust than the Menasce-Huntz algorithm because more information is held at each site.

Our analysis supports the rather obvious conclusion that robustness is inversely related to its cost. The Tsai-Belford algorithm appears more robust than Obersarch's algorithm, for example, but it maintains larger WF graphs at each site, and is invoked each time a resource is requested, in order that the WF graphs contain sufficient information.

For the example we used to analyze the four algorithms in section 3, the behavior of each of those algorithms in the presence of errors is almost identical. Because our deadlock cycle only involved 2 transactions, those algorithms which are potentially more robust in the presence of larger cycles did not have time to develop their robustness. In other words, for a short deadlock cycle, all the algorithms converged within approximately the same length of time (two or three iterations.) Short cycles of length 2 or 3 are more probable in existing applications, so all the above algorithms are approximately equally robust in current applications. In future applications (information utility programs, for example), however, we expect a much higher probability of more complex deadlock cycles, which will require a more robust DDA. Conversely, however, as the number of transactions (and sites increases, it will be important to use a minimum cost DDA.

take it
Work is currently in progress on a new robust distributed deadlock
detection algorithm.
References


INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93940

Office of Research Administration 1
Code 012A
Naval Postgraduate School
Monterey, CA 93940

Chairman, Code 52Bz 30
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Dushan Badal, Code 52Zd 5
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Robert B. Grafton 1
Office of Naval Research
Code 437
800 N. Quincy Street
Arlington, VA 22217

David W. Mizell 2
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106

Vinton G. Cerf 1
DARPA/IPTO
1400 Wilson Blvd
Arlington, VA 22209

CDR R. Ohlander 1
DARPA
1400 Wilson Blvd
Arlington, VA 22209

Col. D. Adams 1
DARPA
1400 Wilson Blvd
Arlington, VA 22209