JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 22, 44–59 (1994)

On Characterization and Correctness of Distributed Deadlock Detection

AJAY D. KSHEMKALYANI*,† AND MUKESH SINGHAL‡

*IBM Corporation, P.O. Box 12195, Research Triangle Park, North Carolina 27709; and ‡Department of Computer and Information Science, The Ohio State University, 2036 Neil Avenue, Columbus, Ohio 43210

Distributed deadlock detection requires identifying the presence of certain properties in the global state of distributed systems. Distributed deadlock detection is complicated due to the lack of both global memory and a common physical clock, and due to unpredictable message delays. We characterize the formation and detection of distributed deadlocks in terms of the contents of local memory of distributed nodes/sites. We describe how the interaction between deadlock detection and deadlock resolution can lead to the detection of false deadlocks that are impossible to avoid due to inherent system limitations. We define shadow, phantom, and pseudo deadlocks in the proposed framework. We give examples of existing incorrect deadlock detection algorithms to illustrate how they violate the developed requirements for distributed deadlock detection. The characterization provides an insight into the properties of distributed deadlocks, expresses inherent limitations of distributed deadlock detection, and yields new correctness criteria for distributed deadlock detection algorithms. © 1994 Academic Press, Inc.

1. INTRODUCTION

The problem of deadlocks in an important, fundamental problem in the design of concurrent systems. Though the problem of deadlocks is well understood in shared-memory systems [8, 9], it remains a notoriously difficult problem in distributed systems [31]. Over the last decade, considerable attention has been focused on the design of algorithms for deadlock detection in distributed systems [31]. However, properties of distributed deadlocks are not yet well understood, and a fundamental question about the definition of a distributed deadlock still remains unanswered. This is why many of the proposed algorithms [7, 14, 33] have been shown to be incorrect—they either do not detect all existing deadlocks or detect false deadlocks.

For example, the classical path-pushing algorithm [25] is incorrect because it fails to take action when information about deadlock becomes available [12]; the probe-based algorithm of Sinha and Natarajan [33] is incorrect because it incorrectly discards relevant probes (causing undetected deadlocks) and retains outdated probes (causing detection of false deadlocks) [7]; the two-phase algorithm of [14] does not distinguish between two wait-for dependency edges between the same resource and process caused by a transaction.

We identify the following as the main causes of the astonishing failure of so many distributed deadlock detection algorithms: first, there is no concept of global time in distributed systems. The definition of distributed deadlocks does not have any temporal specifications. Many existing algorithms are wrong because they do not consider the causality relation [21] among the events at different sites, leading to the inclusion of inconsistent states in their reasoning. This complicates not only developing correct algorithms but also proving their correctness formally. Second, most of the algorithms are based on ad hoc methods in the sense that they take the definition of deadlock from a centralized (shared memory) system and try to extend it to distributed systems. Third, stability of deadlocks is not a correct assumption because detected deadlocks are promptly resolved. A site can independently resolve a deadlock and other sites may learn about it after an arbitrary and random delay. This specially creates problems in the multiple request model of deadlocks because some processes may be common to many deadlock cycles. The deadlock resolution process interacts with deadlock detection in a complex and unpredictable manner [31].

We believe that a distributed deadlock should be defined more rigorously in terms of a causal relationship among events at processes/sites. The conventional definition of deadlocks, viz., "a cycle in a distributed transaction WFG (wait-for graph)," is not appropriate for distributed deadlock detection because in distributed systems no site has current and consistent global state or knowledge of the system. There is a possibility that a global cycle of which different segments existed at non-overlapping time intervals is reported.

In this paper, we give a characterization of distributed deadlocks in terms of the contents of distributed memory and construct a definition of distributed deadlock that explicitly uses virtual/logical global time [10, 11, 23]. The characterization is presented from the points of view of a global and a non-global observer in terms of locally ac-

† At the time this work was performed, this author was with the Department of Computer and Information Science, The Ohio State University, 2036 Neil Avenue Mall, Columbus, OH 43210.

0743-7315/94 $6.00
Copyright © 1994 by Academic Press, Inc.
All rights of reproduction in any form reserved.

cessible variables and timestamps. A global observer has an instantaneous view of the system state with access to shared memory and a common clock. A local observer has to deal with a distributed memory and does not have an instantaneous view of the system state. The characterization provides an understanding of the detection and resolution of distributed deadlocks, and provides new criteria for the correctness of distributed deadlock detection algorithms. This is the first characterization of distributed deadlocks [18] using causality and global virtual time.

The following request models have been proposed in the literature [2, 4, 16]: single-request, AND request, OR request, and generalized request models. The characterization and definition of distributed deadlocks presented in this paper is applicable to the single-request and AND request models, though much of the discussion of the building blocks of deadlock is applicable to other request models, too. In the AND request model, a process issues multiple requests and remains blocked until it receives a reply to each of its outstanding requests. The single-request model is a special case of the AND model in which a process can have a single outstanding request at a time.

The work presented in this paper has appeared in [18, 19]. Deadlock detection for real-time distributed systems is studied in [30]. A theory of deadlocks that is based on an axiomatization of the process-resource model and that allows only the single-request model is given in [34].

The rest of the paper is organized as follows: in Section 2, we introduce definitions and preliminary concepts. In Section 3, we give a characterization of distributed deadlocks for a global and a nonglobal observer. Section 4 examines the interaction between distributed deadlock formation, detection, and resolution. In Section 5, we define various types of false deadlocks. Section 6 examines the problem of deadlocks in a restricted model. In Section 7, we present correctness criteria for detection of distributed deadlocks. Section 8 contains a summary and concluding remarks.

2. PRELIMINARIES

2.1. System Model

The system is a network of N nodes with a logical channel from every node to each other node. Each node represents a site and it is assumed without loss of generality that each site in the system has one process running on it. Hence, nodes are synonymous to processes. Messages are delivered reliably but not necessarily in the order sent.

A node in the system can be a process or a resource manager (which manages a single resource). Computation messages in the system include *request* messages and *reply* messages. A process sends a request to another process and gets blocked waiting for a reply (i.e., a response to its request, which is a consumable resource). A process sends a request to a resource manager and remains blocked until a reply (i.e., permission to access the resource) is received from the resource manager. The processes follow the AND request model.

The above system model can model database resource deadlocks. In a database system, a process can send requests only to resource managers and not to other processes. A resource manager sends a reply to a process when it can grant the process an access to the requested resource. A process should release the resource after using it. Because the model does not use a release message, the return of the resource by the process to the resource manager can be explicitly modeled as follows: the reply from the resource manager to the process, granting the process access to the resource, implicitly carries a request by the resource manager for the release of the resource by the process. When the process is ready to release the resource, it sends a reply to the resource manager. The effect of this modeling is that the resource manager behaves as an active system node that follows the single-request model. An algorithm for detecting database deadlocks in the AND request model, which is based on the characterization presented in this paper, is given in [19].

Events in the system are message send events, message receive events, and internal events. A system run associates each node i with a totally ordered set C_i of events. Let $C = \cup C_i$ be the possibly infinite set of all events. Some definitions from the literature are presented next.

Causality [21]: For events in C, the causality (or happens before) relation $<$ is defined as the smallest transitive relation satisfying the following two conditions: (i) if $a, b \in C_i$ and a occurs before b, then $a < b$; (ii) if a is the sending of a message and b is the receipt of the same message, then $a < b$.

A run of a computation [5]: $C = (\cup C_i)$ is a run of a computation if $(C, <)$ is acyclic.

It is observed that all real computations are acyclic and $(C, <)$ is a strict partial order.

A partial run of a computation [5, 23]: Any $C' = \cup C'_i \subseteq C$ is a partial run of a computation if C' is left-closed[1] under $<$.

For a set S of totally ordered elements, $max(S)$ is defined to be the maximum element of the set S.

A consistent cut of a computation [27]: A consistent cut, *CCut*, in any computation is a set of (local states at processes corresponding to) events $\cup_{(\forall i)}\{max(C'_i)\}$ for any partial run C' of the computation.

2.2. Vector Timestamps

Due to the absence of a global clock, the existence of a deadlock must be qualified by the local time at which each member of the cycle deadlocks. The local time on a node is given by a vector clock [10, 11, 23] which exactly

[1] For any $a, b \in C$, $b \in C' \wedge a < b \Rightarrow a \in C'$.

captures the partial ordering among events in the computation and is the best approximation to the latest state in the system.

The logical time is defined to be a vector of length N, the number of nodes. The logical time at node i is $\mathcal{T}_i$ and the timestamp of a message msg is $msg.T$. ($\mathcal{T}_i[j]$ and $msg.T[j]$, respectively, denote the jth components of these time vectors.) Events are assigned timestamps T_i which are the clock values $\mathcal{T}_i$ when the events occur. Henceforth, events will be referred to by their unique timestamps; this gives a clean notation.[2] The logical time at a node evolves as follows:

(a) When an internal event or a message send occurs on i, $\mathcal{T}_i[i] := \mathcal{T}_i[i] + 1$.
(b) When i receives a message msg, then $\forall j$ do,

$$\text{if } j = i, \text{ then } \mathcal{T}_i[j] := \mathcal{T}_i[j] + 1,$$
$$\text{else } \mathcal{T}_i[j] := max(\mathcal{T}_i[j], msg.T[j]).$$

Thus, the jth component of the time vector at a node reflects the highest value of the jth component of all timestamped messages it has received. Note that $\mathcal{T}_i[i]$ reflects only the local activities at node i and $\mathcal{T}_i[j]$ reflects what node i knows about the local timestamp (i.e., activities) of node j. Thus, time vector $\mathcal{T}_i$ reflects what node i knows of the latest state (local time) of all other nodes. $abs(T_i)$ refers to the absolute real-time at which the event represented by T_i occurred.

An ordering relation "$<$" between timestamps is defined as follows: $T_i < T_j$ iff $(\forall k)$, $T_i[k] \leq T_j[k] \wedge T_i \neq T_j$. $T_i < T_j$ iff the event at T_i happened before [21] the event at T_j. If $T_i \not< T_j$ and $T_j \not< T_i$, then T_i and T_j are concurrent and are denoted by T_i *co* T_j. Thus, the logical clock proposed in [10, 23] defines the exact causal order as a partial order which is weaker than the total order of Lamport's clocks [21].

Though vector timestamps appear to be large, there are efficient schemes to reduce the size of vector timestamps that are transmitted [24, 32]. Vector clocks are used in the proposed characterization because they are capable of capturing the partial ordering among distributed events[3] which is necessary for observing distributed deadlocks. Thus, vector timestamps allow us to give a complete characterization of distributed deadlocks. A distributed deadlock detection algorithm does not necessarily have to use vector timestamps in its implementation. The only effect of not using vector timestamps will be that false deadlocks may be detected because temporal ordering of events cannot be captured (see Section 5 for more details).

[2] With this convention, reference to a vector timestamp does not require access to the entire vector, unless the components are explicitly referenced.

[3] Charron-Bost [6] has shown that the minimum length of the vector timestamp to capture the partial ordering of events in a distributed system is N, the number of nodes.

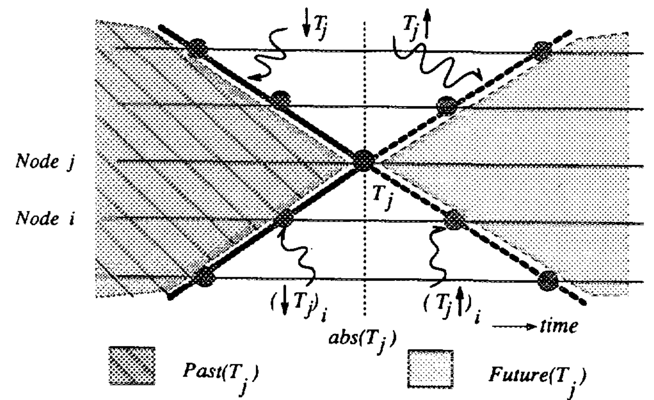


FIG. 1. Past and future cones of T_j.

2.3. Past and Future Cones

An event T_j could have been affected only by all events T_i that satisfy $T_i < T_j$. All such events T_i belong to the past of T_j [22, 23]. This and the subsequent definitions are illustrated in Fig. 1. Let $Past(T_j)$ be the set of all events that happened causally before T_j. This is the set of all events in the past of T_j in the computation. In Fig. 1, $Past(T_j)$ can be viewed as the past cone. Let $Past_i(T_j)$ be the set of those events of $Past(T_j)$ that are on node i. $Past_i(T_j)$ is totally ordered by relation "$<$" and its maximal element is $max(Past_i(T_j))$. $max(Past_i(T_j))$ is the latest event on node i that has affected T_j. Let $\downarrow T_j = \cup_{(\forall i)}\{max(Past_i(T_j))\}$ in the computation. Note that $\downarrow T_j$ consists of the latest event on each node that affects T_j, and is referred to as the *surface of the past cone* of T_j [22, 23]. The system state along $\downarrow T_j$ is a consistent cut [27] because $Past(T_j)$ is left-closed. Let $(\downarrow T_j)_i$ denote the element of $\downarrow T_j$ that is on node i. It is obvious that $(\downarrow T_j)_i[i] = T_j[i]$ for all i. Similar to the past is the future of T_j representing all the events T_i that are causally affected by T_j, giving rise to the definitions of $Future(T_j)$ and the surface of the future cone $T_j\uparrow$. (The notations $\downarrow T_j$ and $T_j\uparrow$ are due to Charron-Bost [5].) In Fig. 1, events in the unshaded region occur concurrently with T_j.

2.4. Intervals on a Node

This section introduces terminology that is specific to characterizing distributed deadlocks and proving the correctness of distributed deadlock detection algorithms.

A node can be either in a blocked or an unblocked state. A node i blocks when it sends out a request and remains blocked until its request is satisfied. This dependency is denoted by adding a wait-for edge to the WFG. All valid inward dependencies on i and outward dependencies from i are stored in local set variables $in(i)$ and $out(i)$, respectively. Let T_out_i and T_in_i be the timestamps when node i sends a request (establishes an outward dependency) and receives a current request (establishes an inward dependency), respectively.

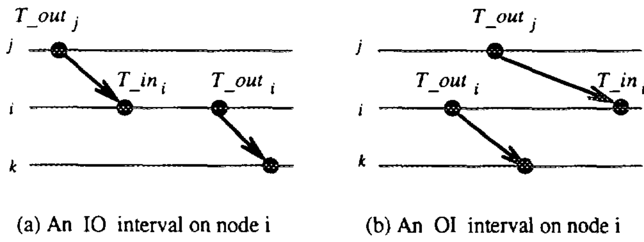


FIG. 2. IO and OI intervals.

Define the *interval* for the latest outward dependency and an inward dependency as the duration from the establishment of the earlier until the establishment of the later one. An interval is *completed* when both dependencies of the interval are established and the first dependency is valid as far as the node can discern, at the time the second dependency gets established. There are two types of intervals, shown in Figs. 2a and 2b, depending upon whether the inward dependency is established before the outward dependency or vice-versa. The former interval is referred to as an *IO interval,* the latter as an *OI interval*.

The completion of an interval at a node signifies the possible participation of the node in a distributed deadlock. (Note that a node can be involved in a deadlock even before an OI interval completes successfully.) An interval begins at node i whenever one of the following two cases arises: (a) i is not blocked and receives a request from node j, or (b) i gets blocked on node k. For the two cases (a) and (b) in which an interval starts, the interval completes if (a) i blocks and the request from j is still pending, and (b) i receives a request from j and its request to k is still outstanding as far as it can discern, respectively.

3. CHARACTERIZATION OF DISTRIBUTED DEADLOCKS

Intuitively, a deadlock comprises of a wrap-around sequence of nodes, each of which stays blocked indefinitely on the next member of the sequence until a deadlock resolution can be performed. However, this definition is incomplete in a distributed system. This is because in a distributed system, a deadlock must be defined in terms of what a node can observe and infer about the global system state, not in terms of what one can ascertain at some instant in global time. A global deadlock must be characterized in terms of the contents of distributed memory and some causal relationship among time instants of local observations. One must specify the exact events in the system as seen by a local observer that lead to a deadlock. Thus, one seeks the relationship among these events (events of deadlock formation and observation by a local observer) for detecting a deadlock.

This section first examines the building blocks that lead to a deadlock, as seen by a global observer. We give a definition of deadlocks and justify it by reasoning about the building blocks of deadlock as seen by the global observer. This gives an insight into the formation of deadlocks in a distributed system. Due to the nonexistence of a global observer, attention needs to be focused on characterizing distributed deadlock in terms of what each node knows of the rest of the system. The building blocks of deadlock are recast as what is observed by a nonglobal observer. The definition of distributed deadlock is adapted and the changes are justified by explaining the impossibility of an instantaneous view of the distributed system.

The definitions of the building blocks of a deadlock and a deadlock have two parts: (I) the conditions that must be satisfied by the blocking events; (II) the conditions that must be satisfied when the system is observed. The above two types of conditions are expressed in terms of timestamps and local variables at the involved nodes.

This section also examines the interaction between deadlock detection and resolution. It turns out that a detected deadlock is assured to exist at the time of detection only in a restricted model.

We next present two axioms which describe the blocking and unblocking of a node in a computation.

Axiom 1. A node blocks when it sends a request and it does not send any computation messages until it gets unblocked.

Therefore, no other node gets a computation message from a blocked node (other than the request on which it blocks) as long as it stays blocked. In the AND request model, a process atomically sends requests with the same timestamp and gets blocked.

Axiom 2. A blocked node can get unblocked if: (i) its requests are satisfied during the normal course of processing (called the "normal mode") or (ii) it spontaneously withdraws its requests or its requests are satisfied due to resolution of a deadlock of which it is a part (called the "abnormal mode").

When a deadlock exists, each involved node must be blocked on its successor. Moreover, it must stay blocked until a resolution is done.

3.1. A Global Observer's View

A global observer has instantaneous access to the state of all nodes and consequently, it (i) can order all events on a scalar time scale, (ii) has access to global time, and (iii) can use global time in reasoning. In a computation, at any global time t, there could have existed a sequence $\langle T_{i_1}, T_{i_2}, \ldots, T_{i_n}\rangle$ of events on nodes $i_j \in P = \{i_1, i_2, \ldots, i_n\}$ that satisfy some combination of the following conditions (henceforth, $i_j \in \{i_1, i_2, \ldots, i_n\}$):

- **(C1)** Each T_{i_j} is the latest event $T_out_{i_j}$ at or before t, when i_j got blocked on its successor i_{j+1}.
- **(C2)** A node i_j (except for $j = 1$) has received the request (on which i_{j-1} got blocked) from i_{j-1} before $T_out_{i_j}$.

(C3) Each node i_j (except for $j = 1$) has not replied to the request on which node i_{j-1} is waiting on it until $T_out_{i_j}$.

(C4) A node i_j (except for $j = 1$) has not received any message for which i_{j-1}'s component of the timestamp is greater than $T_out_{i_{j-1}}[i_{j-1}]$, until $T_out_{i_j}$.

(C5) Each node i_j is blocked on i_{j+1} at time t.

The notions of segments and paths which are the building blocks of deadlocks are introduced. Both signify a sequence of nodes in which each node is blocked on its successor in the sequence. The timestamps at which nodes with successive events in the sequence block satisfy a weaker relationship in a path than in a segment.

Segments

DEFINITION 1. A *segment* at global time t is a sequence of events $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ at nodes $i_1, i_2, \ldots, i_n$, satisfying the following conditions:
(I) (C1), (C2), (C3), and (C4).
/* conditions on blocking events. */
(II) (C5).
/* conditions when the system is observed. */

Every event in a segment denotes a node blocked on its successor in the segment at the event of observation[4] (condition (C5)). Moreover, when node i_j blocks (as per (C1)), its predecessor's request has been received (condition (C2)) and is pending because i_j has not responded to it (condition (C3)), nor is i_j aware that the predecessor has withdrawn its request (condition (C4)). A segment denotes a sequence of blocked nodes with the above properties. In this sequence, the dependencies of the nodes on their successors are created sequentially. A segment can be considered as a basic sequence because it represents a linear growth of wait-for dependencies that is the same as the order in the sequence. Given a segment at global time t, every node preceding a given node i_j on the segment is blocked directly or transitively on i_j at t. The length of a segment increases by one when the successor of the last node on the segment blocks (condition (C1)) while satisfying conditions (C2), (C3), and (C4). Along a segment, a node (except the first node) blocks after it has received the request from its predecessor (condition (C2)). This implies that $T_out_{i_j} > T_out_{i_{j-1}}$ for $j > 1$. It can be inferred that the first event $T_out_{i_1}$ in a segment signifies a potential OI interval and the rest of the events signify successfully completed IO intervals.

In a distributed computation, it may happen that $T_out_{i_j} \not\leq T_out_{i_{j+1}}$ and there exists a dependency from i_j to i_{j+1} which is blocked at $T_out_{i_{j+1}}$, (and transitively to the successors of i_{j+1} if any). So the notion of *path* is introduced to characterize such a sequence of blocked nodes.

[4] Henceforth, a reference to "a node on a segment/path" will mean "a node with an event on a segment/path."

Paths

DEFINITION 2. A *path* at global time t is a sequence of events $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ at nodes $i_1, i_2, \ldots, i_n$, satisfying the following conditions:
(I) (C1) and ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))).
/* conditions on blocking events. */
(II) (C5).
/* conditions when the system is observed. */

Every node on a path is blocked on its successor at the event of observation (condition (C5)). However, unlike a segment, a node i_j with an event on a path may not have received i_{j-1}'s request when i_j blocks at $T_out_{i_j}$ (an OI interval at node i_j.) Thus, on a path, the first event in the sequence denotes a potential OI interval whereas the others may denote either IO or OI completed intervals. (Note that for a segment, all the nodes except the first have only IO completed intervals.) A path can be expressed as a concatenation of several segments.

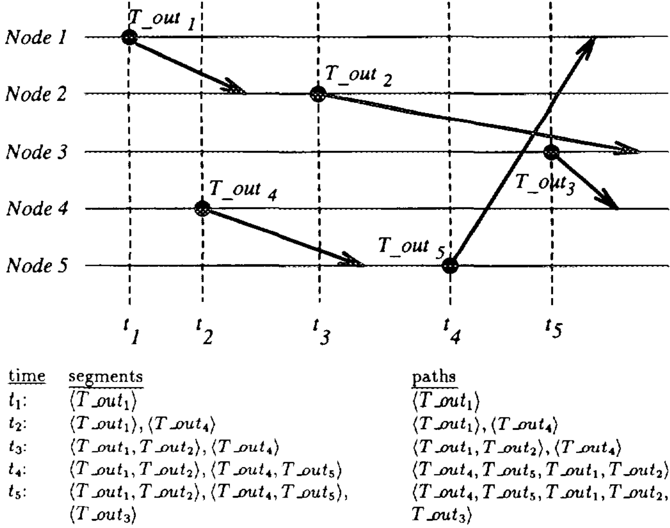
A path at global time t signifies that every node on the path preceding a given node i_j ($j > 1$) on the path is blocked directly or transitively on i_j at t.

Conditions ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))) [referred to as *consecutive blocking*] and (C1) indicate that node i_{j-1} is blocked at $T_out_{i_{j-1}}$ on i_j which blocked at $T_out_{i_j}$, and all nodes on the path will remain blocked forever unless either the last node on the path receives a reply from its successor or some node on the path withdraws its request. However, while observing the system to detect a path, it must be ensured not only that the latest available state of each node is observed, but also that in this state, all nodes that are believed to be consecutively blocked are still blocked. This is expressed in condition (C5) [referred to as *global consistency of blocking at latest observation events*]. Note that condition ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))) asserts the blocking of nodes only at instants $T_out_{i_j}$. Condition (C1) relates these $T_out_{i_j}$ to the latest available observable state (Condition (C5)) by specifying that each $T_out_{i_j}$ is the latest event at which i_j blocked before i_j was observed.

Note that the condition (C2) $\Rightarrow$ (C4) is true if (C1) and (C5) are true, but this condition is explicitly stated because i_j can verify it at $T_out_{i_j}$ for the latest request received from i_{j-1} until $T_out_{i_j}$. For consecutive blocking, strictly one could require the condition $\overline{(C2)}$ $\Rightarrow$ (C4) as well but this condition is not explicitly stated because it cannot be evaluated at $T_out_{i_j}$. Note that this condition is true if (C1) and (C5) are true.

Conditions for Deadlocks

A segment/path continues to exist so long as all the involved nodes remain blocked. This property is of interest for special paths that denote deadlocks. This paper focuses on deadlocks in the single-request and AND request models only, where a cycle in the WFG is the necessary and sufficient condition for a deadlock. In the OR request model, a knot in the WFG is the necessary



time	segments	paths
t_1:	$\langle T_out_1 \rangle$	$\langle T_out_1 \rangle$
t_2:	$\langle T_out_1 \rangle, \langle T_out_4 \rangle$	$\langle T_out_1 \rangle, \langle T_out_4 \rangle$
t_3:	$\langle T_out_1, T_out_2 \rangle, \langle T_out_4 \rangle$	$\langle T_out_1, T_out_2 \rangle, \langle T_out_4 \rangle$
t_4:	$\langle T_out_1, T_out_2 \rangle, \langle T_out_4, T_out_5 \rangle$	$\langle T_out_4, T_out_5, T_out_1, T_out_2 \rangle$
t_5:	$\langle T_out_1, T_out_2 \rangle, \langle T_out_4, T_out_5 \rangle, \langle T_out_3 \rangle$	$\langle T_out_4, T_out_5, T_out_1, T_out_2, T_out_3 \rangle$

FIG. 3. Formation of a closed path of five nodes.

and sufficient condition for deadlock [2, 4, 16]. Much of the discussion about paths and segments is valid, although with a different semantics, for the OR request and generalized request models. However, the conditions for deadlock in these models need to be formulated differently.

DEFINITION 3. A path $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n} \rangle$ is a *closed path* iff node i_n is blocked at $T_out_{i_n}$ on i_1.

Path $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_{n-1}} \rangle$ closes at the instant that the successor i_n of the last node i_{n-1} on the path blocks at $T_out_{i_n}$ on the first node i_1 on the path (the path concatenates with itself.) Figure 3 shows a closed path with five nodes. The legend shows the formation of the closed path by listing the segments and paths as seen by a global observer at successive instants in absolute-time when the participating nodes block.

DEFINITION 4. A closed path observed at global time t denotes a deadlock at instant t. (i.e., each node on the path will remain blocked on its successor forever unless a node on the path unblocks as per Axiom 2(ii).)

Explanation. A node i_j on a closed path is blocked on its successor at $T_out_{i_j}$. At t, each node on the closed path is blocked on its successor and no node on the path can get unblocked as per Axiom 2(i). There exists a deadlock. Assuming no node spontaneously withdraws its request or is aborted (the system does not obey Axiom 2(ii)), this state will persist forever. The smallest value t can take is $max_{(\forall i_j)}(abs(T_out_{i_j}))$.

The above analysis assumed the existence of a global observer. A global observer with instantaneous access to all node states has precise information of the deadlock state and mimics access to a common memory and global time. Though a global observer is unrealistic, the above analysis provides an insight into distributed deadlocks which will help us in understanding distributed deadlocks in the absence of a global observer.

3.2. Coping with a Distributed View

Since a global observer is not realizable, one has to deal with local observers. This means that one has to confine one's reasoning to what individual nodes know of the system. Without loss of generality, it can be assumed that there is a unique event T_detect_x at node x in the system that observes a deadlock (closed path).

A node cannot make any assertions about the system state at absolute time t. The most it can deduce about the system is what it observes from all the messages it has received until the event of observation. A node can observe all the events that causally happened before the observation event. These observable events identify the computation prefix $Past(T_detect_x)$ which is pictorially represented by the past cone [22, 23] shown in Fig. 1. Thus, the vertical observation line at global time t becomes the surface of a cone with vertex at T_detect_x due to the absence of common memory and common clock, and due to non-zero message delay in the system. $\downarrow T_detect_x$ is a consistent cut [27] and node x can draw valid conclusions from events in $Past(T_detect_x)$.

In this framework, we have to adapt the definitions of conditions (C1) and (C5) that a sequence $\langle T_{i_1}, T_{i_2}, \ldots, T_{i_n} \rangle$ of events on nodes $i_j \in P = \{i_1, i_2, \ldots, i_n\}$ must satisfy:

(C1') Each T_{i_j} is the latest time $T_out_{i_j}$ at or before $(\downarrow T_detect_x)_{i_j}$ when i_j got blocked on its successor i_{j+1}.

(C5') Each node i_j is blocked at $(\downarrow T_detect_x)_{i_j}$ on i_{j+1}.

The definitions of segment and path are the same as before except that absolute time references are changed to vector time references.

Segments

DEFINITION 1'. A *segment* at time T_detect_x is a sequence of events $\langle T_out_{i_1}, t_out_{i_2}, \ldots, T_out_{i_n} \rangle$ at nodes $i_1, i_2, \ldots, i_n$, satisfying the following conditions:

(I) (C1'), (C2), (C3), and (C4).
/* conditions on blocking events. */

(II) (C5').
/* conditions when the system is observed. */

Given a segment at time $T_detect_{i_k}$ where i_k, $1 < k \leq n$, is a node on the segment, every node i_j preceding i_k is blocked directly or transitively on i_k at $T_out_{i_j}$. Node i_k can get to know this because $T_out_{i_j} > T_out_{i_{j-1}}$ for nodes i_j $(j > 1)$ on the segment. Hence for nodes i_j, $1 \leq j < k$ on the segment, $T_out_{i_j} \leq (\downarrow T_out_{i_k})_{i_j}$, i.e., $T_out_{i_j}$ belongs to the past of every other event that occurs after it in its segment.

Paths

DEFINITION 2′. A *path* at time T_detect_x is a sequence of events $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ at nodes $i_1, i_2, \ldots, i_n$, satisfying the following conditions:

(I) (C1′) and ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))).
/* conditions on blocking events. */

(II) (C5′).
/* conditions when the system is observed. */

If the outward dependencies of some two arbitrary nodes do not lie on the same segment on a path, neither node may know that the other is blocked since the past cone of neither event at which the dependency is formed necessarily includes the other. Node i_j can get to know that its predecessors on its segment of the path are blocked on it directly or transitively. Note that if a node i_j on a path at T_detect_x does not advance its local clock until it unblocks, then $T_out_{i_j}[i_j] = T_detect_x[i_j]$.

3.2.1. Conditions for Paths Revisited

We now express the conditions for a path in terms of timestamps and local variables at a node.

Condition (C2) $\Rightarrow$ ((C3) $\wedge$ (C4))

LEMMA 1. *A sequence of blocking events* $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ *where each* i_j *is blocked on* i_{j+1} *satisfies* (C2) $\Rightarrow$ ((C3) $\wedge$ (C4)) *iff the nodes* $i_j \in \{i_1, i_2, \ldots, i_n\}$ *that have IO intervals satisfy the following* consecutive blocking *conditions*:

(1) $T_out_{i_{j-1}}[i_{j-1}] = T_in_{i_j}[i_{j-1}] = T_out_{i_j}[i_{j-1}]$

(2) $i_{j-1} \in in(i_j)$ *when the node* i_j *blocks.*

Proof. Part 1 (*Sufficiency*): $T_in_{i_j}[i_{j-1}] = T_out_{i_{j-1}}[i_{j-1}]$ implies that when the IO interval starts on node i_j, the latest i_j knows of i_{j-1}'s status is that it has made the request at $T_out_{i_{j-1}}[i_{j-1}]$. i_j adds i_{j-1} to $in(i_j)$. $T_out_{i_j}[i_{j-1}] = T_out_{i_{j-1}}[i_{j-1}]$ implies that i_j has not received a message that was sent directly or transitively indicating that i_{j-1} has sent a message after i_{j-1} blocked at $T_out_{i_{j-1}}$. This proves (C2) $\Rightarrow$ (C4). $i_{j-1} \in in(i)$ at the end of the interval implies that some request made by i_{j-1} is valid. Since i_j has not received any request from i_{j-1} issued after the one issued at $T_out_{i_{j-1}}$, the request that is valid at $T_out_{i_j}$ is the same one as that at $T_in_{i_j}$. Hence, i_j has not replied to the request on which its predecessor is blocked at $T_out_{i_{j-1}}$. This proves (C2) $\Rightarrow$ (C3).

Part 2 (*Necessity*): (C2) $\Rightarrow$ ((C3) $\wedge$ (C4)). The request received by i_j from i_{j-1} is valid when it is received because (C4) holds. On receiving the request, $T_in_{i_j}[i_{j-1}] = T_out_{i_{j-1}}[i_{j-1}]$ by the way the clocks are operated, and i_{j-1} is added to $in(i_j)$. Because (C4) holds, $T_out_{i_j}[i_{j-1}] = T_in_{i_j}[i_{j-1}]$. This proves (1). Since i_j does not respond to the request until it blocks (because of (C3)), it does not make the request invalid at $T_out_{i_j}$ by a local action, and because of (C4), i_j does not know the request is made invalid by an external action. So the request is valid at $T_out_{i_j}$ and $i_{j-1} \in in(i_j)$ at $T_out_{i_j}$. This proves (2). The above reasoning is valid throughout the IO interval on i_j, hence $i_{j-1} \in in(i_j)$ throughout the IO interval. ■

Condition (C2) $\Rightarrow$ ((C3) $\wedge$ (C4)) satisfied by events in $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ signifies that for a node i_j with a completed IO interval, its predecessor i_{j-1} is blocked consistently on it, (i.e., the blocked status and timestamp $T_out_{i_{j-1}}$ at which i_{j-1} blocked do not contradict i_j's knowledge of i_{j-1}). This is an assertion on i_j at $T_out_{i_j}$ and on i_{j-1} at $T_out_{i_{j-1}}$. These local assertions at each $T_out_{i_j}$ do not imply that $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ forms a path. To observe the existence of a path, the system must be observed at T_detect_x along the consistent cut $\downarrow T_detect_x$ and it must be ensured in addition that: (a) each node i_j blocked at $T_out_{i_j}$ is blocked at $(\downarrow T_detect_x)_{i_j}$ (C5′), and (b) events in the sequence $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$, on which (C2), (C3), and (C4) are specified, are related to $\downarrow T_detect_x$ as per [C1′], i.e., each node i_j blocked at $T_out_{i_j}$ in the sequence must *remain blocked* until $(\downarrow T_detect_x)_{i_j}$.

Note that i_j may infer at $T_out_{i_j}$ using (C2) $\Rightarrow$ (C3) $\wedge$ (C4) that it has a completed IO interval for the latest request received from i_{j-1}, even though i_{j-1} may have unblocked and sent a newer request to i_j, signifying an OI interval at i_j in a path observed at T_detect_x. This occurs because i_j cannot ensure (C1′) with respect to T_detect_x when it receives the latest request from i_{j-1}. Even so, this inference is useful for distributed deadlock detection.

Condition (C5′)

Condition (C5′) pertains to observing a set of blocked nodes consistently along $\downarrow T_detect_x$. The condition that a set of observed events $\{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ lie on a consistent cut is stated by Mattern [23],

$$sup(T_{i_1}, T_{i_2}, \ldots, T_{i_n}) = [T_{i_1}[i_1], T_{i_2}[i_2], \ldots, T_{i_n}[i_n]]$$

where $sup(T_{i_1}, T_{i_2}, \ldots, T_{i_n}) = T | (\forall i_j), (T[i_j] = max(T_{i_1}[i_j], \ldots, T_{i_n}[i_j]))$.

LEMMA 2. *A set of events* $\{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ *observed at* T_detect_x *satisfies* (C5′) *iff the events in the set satisfy the following global consistency conditions*:

(1) $sup(T_{i_1}, T_{i_2}, \ldots, T_{i_n}) = [T_{i_1}[i_1], T_{i_2}[i_2], \ldots, T_{i_n}[i_n]]$, *and* $\forall i_j$ *in the set,* $T_{i_j}[i_j] = T_detect_x[i_j]$

(2) $(\forall i_j) | T_{i_j}$ *is in the set,* i_j *is blocked at* T_{i_j} *on* i_{j+1}.

Proof. Part 1 (*Necessity*): If $\{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ satisfies (C5′) when observed at T_detect_x, each T_{i_j} in the set lies on $\downarrow T_detect_x$. W show that (1) and (2) are satisfied. (1) $T_{i_j}[i_j] = (\downarrow T_detect_x)_{i_j}[i_j]$ and therefore $[T_{i_1}[i_1, T_{i_2}[i_2], \ldots, T_{i_n}[i_n]] = T_detect_x$. Since events on $\downarrow T_detect_x$ lie on a consistent cut, the first equality follows [23]. (2) Clearly, i_j is blocked at each T_{i_j} on i_{j+1} as per (C5′).

Part 2 (*Sufficiency*): It is shown that $T_{i_j}[i_j] = (\downarrow T_detect_x)_{i_j}[i_j]$ when $\{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ satisfies (1) and (2). If

$T_{i_j}[i_j] < (\downarrow T_detect_x)_{i_j}[i_j]$, then $T_{i_j}[i_j] < T_detect_x[i_j]$, a contradiction. If $T_{i_j}[i_j] > (\downarrow T_detect_x)_{i_j}[i_j]$, then $T_{i_j}[i_j] > T_detect_x[i_j]$, a contradiction. It can be concluded that T_{i_j} lies on $\downarrow T_detect_x$ and $\{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ is a consistent cut. Clearly, node i_j is also blocked at T_{i_j} on i_{j+1}. So (C5′) is satisfied. ■

Note that vector timestamps are not needed for (C2) ⇒ ((C3) ∧ (C4)) but are needed for (C5′).

3.2.2. Comments on "Event T_detect_x"

We examine the relation between an observation event T_detect_x and the consistent cuts that it can observe. This helps in determining what global states can be and should be observed.

A consistent cut $CCut = \{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ has a timestamp $[T_{i_1}[i_1], T_{i_2}[i_2], \ldots, T_{i_n}[i_n]]$ as shown by Mattern [23]. Charron-Bost shows that in a run of a computation C, every consistent cut is generated by a unique antichain[5] in $(C, <)$ [5]. Moreover, every antichain of size ≥2 in C generates a unique consistent cut with timestamp T_detect_x such that the supremum of the timestamps of the events in the anti-chain is T_detect_x [5]. Such a timestamp does not correspond to a real event on nodes $i_1, i_2, \ldots, i_n$. Only antichains of size 1 in C, i.e., individual events in C, generate a unique consistent cut with the timestamp of a real event, namely, the event itself; the other events on that consistent cut belong to the past of that event.

ASSERTION 1. *For any event T_detect_x, there exists a* **unique observable** *consistent cut that is exactly $\downarrow T_detect_x$.*

Justification. Let $CCut = \{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$ be the unique consistent cut identified by and containing T_detect_x (such a cut exists [5]). For any $T_{i_j} \in CCut$, there are three cases: (1) $T_{i_j}[i_j] < (\downarrow T_detect_x)_{i_j}[i_j]$, in which case T_{i_j} cannot be in $CCut$. (2) $T_{i_j}[i_j] > (\downarrow T_detect_x)_{i_j}[i_j]$, in which case T_{i_j} is not observable at T_detect_x. (3) $T_{i_j}[i_j] = (\downarrow T_detect_x)_{i_j}[i_j]$, in which case T_{i_j} is observable at T_detect_x and can belong to $CCut$. Thus, each element of $\downarrow T_detect_x$ belongs to the unique observable consistent cut $CCut$.

3.2.3. Conditions for Deadlock

THEOREM 1. *There exists a path at T_detect_x iff there exists a set $S = \{i_1, i_2, \ldots, i_n\}$ of nodes such that*

(1) The set of events $(\downarrow T_detect_x)_{i_j}$, *where $i_j \in S$, satisfies (C5′) (global consistency of blocking at latest observation events).*

(2) The sequence of events $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ *identified by (C1′), satisfies (C2) ⇒ (C3) ∧ (C4) (consecutive blocking).*

[5] An *antichain* of a partially ordered set is a subset in which any two elements are incomparable.

Proof. Follows from the definition of path. Lemmas 1 and 2 indicate how to evaluate (C2) ⇒ (C3) ∧ (C4) and (C5′), respectively. ■

COROLLARY 1. *The set of events in the sequence $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ identified in Theorem 1 forms a consistent cut.*

Proof. $\downarrow T_detect_x$ is a consistent cut. No message sent after $(\downarrow T_detect_x)_{i_j}$ reaches node i_k before $(\downarrow T_detect_x)_{i_k}$ and therefore before $T_out_{i_k}$. Also, no message is sent between $T_out_{i_j}$ and $(\downarrow T_detect_x)_{i_j}$. Therefore, no message sent after $T_out_{i_j}$ reaches i_k before $T_out_{i_k}$. The result follows. ■

DEFINITION 4′. A closed path at T_detect_x denotes a deadlock at T_detect_x (i.e., each node on the path will remain blocked on its successor forever unless a node unblocks as per Axiom 2(ii)).

Explanation. By Axioms 1 and 2, a blocked node on the path will not send any message (i.e., will remain blocked) until it receives a reply from its successor on the path or it unblocks in the abnormal mode (Axiom 2(ii)). Consider a node i_h which has an OI interval. i_h stays blocked at least until its successor i_{h+1} sends i_h a reply or i_h unblocks as per Axiom 2(ii). If i_{h+1} has an IO interval, it does not send i_h a reply until i_{h+2} sends i_{h+1} a reply or i_{h+1} unblocks as per Axiom 2(ii). By induction, a node i_k such that i_k is the first node on the next segment in the path is reached. Nodes i_h and i_k have OI intervals and i_h stays blocked at least until i_k sends a reply to its predecessor or some node on the segment from i_h to i_k unblocks as per Axiom 2(ii).

No node i_j on the path has become unblocked until $(\downarrow T_detect_x)_{i_j}$, which is the most information available at instant T_detect_x. If after $(\downarrow T_detect_x)_{i_j}$, no node unblocks as per Axiom 2(ii)), then i_j stays blocked on i_k at least until i_k sends a reply to its predecessor.

Assuming that after $\downarrow T_detect_x$, no node unblocks as per Axiom 2(ii), it can be observed that every node with an OI interval on the path stays blocked at least until the next node with an OI interval on the path sends a reply to its predecessor. By induction, all the nodes with OI intervals on the closed path stay blocked at least until they send a reply to their predecessors; however, a blocked node will never send a reply. Also, a node with an IO interval on the path stays blocked at least until the node (with an OI interval) on the next segment on the path sends a reply to its predecessor.

Hence, all the nodes remain blocked forever, implying that $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ is a deadlock. Note that none of the blocked nodes can have their requests satisfied in the normal mode (case (i) in Axiom 2).

3.2.4. An Inherent Limitation in Observing Deadlocks

A deadlock at T_detect_x is based on what can be observed along the unique consistent cut $\downarrow T_detect_x$

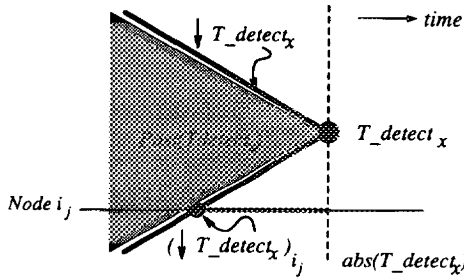


FIG. 4. Past cone of detection event T_detect_x.

whose events lie along the thick line in Fig. 4. However, no assertion can be made about the deadlock status at $abs(T_detect_x)$ because events that occured at any node i_j between $(\downarrow T_detect_x)_{i_j}$ and $abs(T_detect_x)$ cannot be observed at $abs(T_detect_x)$. This is an inherent limitation of a nonglobal observer in a distributed system. There is no way of guaranteeing that at the absolute time when a deadlock is detected, the deadlock indeed exists.

This inherent limitation specially creates a problem when nodes can unblock in the abnormal mode (Axiom 2). This can result in the detection of false deadlocks (i.e., deadlocks that do not exist at the time of detection). For example, node i_j's request can be satisfied in the abnormal mode (Axiom 2(ii)) during a period from $(\downarrow T_detect_x)_{i_j}$ to $abs(T_detect_x)$, as a result of which the deadlock may not persist at $abs(T_detect_x)$. However, it will be observed by node x at $abs(T_detect_x)$.

3.2.5. Inferring if a Set of Observed Events Forms a Deadlock

The problem of detecting distributed deadlocks can be phrased as follows: given a set of events $\{T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\}$, do they form a deadlock for an observer? This is equivalent to sampling each node i_j at $T_out_{i_j}$ and determining if these events form a deadlock. Based on the results obtained so far, we answer this question next.

Given a set of events, the past of all these events can be constructed by doing a union on the pasts of each event [5]. This operation yields a consistent global state because it takes the transitive closure under the happens before relation of all the messages known to be received. Given a set of events $\{T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\}$ reported to a control node x, it can construct $\cup_{(\forall i_j)}(Past(T_out_{i_j}))$ to give $Past(T_detect_x)$. The observer x can now check $Past(T_detect_x)$ for a closed path by using Theorem 1. (Another way of looking at a deadlock detection event is that it lies in the intersection of the future cones of the deadlocked nodes.)

Centralized Detection of Deadlocks

We can use the above result to devise a centralized deadlock detection scheme which works as follows: each node i_j reports its status to a central node x that maintains the WFG along $\downarrow T_detect_x$. The status reported by i_j includes all local variables that are necessary for evaluation of a path. These variables are required in Theorem 1 which uses Lemmas 1 and 2 to evaluate (C2) $\Rightarrow$ ((C3) $\wedge$ (C4)) and (C5′), respectively. Specifically, at the time node i_j blocks, it reports to node x the following variables: (i) $T_out_{i_j}$ $(= T_{i_j})$, (ii) for every other node i_k, the i_kth component of the timestamp $T_in_{i_j}$ when the most recent request from i_k was received, and (iii) for every other node i_k, a boolean value indicating whether currently $i_k \in in(i_j)$. Node i_j also reports its timestamp at the time it gets unblocked.

The central site x works as outlined next: whenever it gets new data, it **(1)** updates its WFG by adding new wait-for edges and deleting old wait-for edges. **(2)** If there is a wrap-around sequence of blocked nodes, it checks if each adjoining pair in the sequence satisfies consecutive blocking ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))). **(3)** If (2) holds and each blocked node i_j in the wrap-around sequence remains blocked until $\downarrow T_detect_x$ ((C5′) and (C1′)), it declares deadlock. A variant of this scheme is given in [17].

3.3. Distributed Deadlock Detection

Distributed deadlock detection can be viewed as detection of a special type of consistent cut (global snapshot [3]), in which each node satisfies consecutive blocking and global consistency of blocking at the latest observation events. For distributed detection of deadlocks, each blocked node should be able to confirm both its incident dependencies, (i.e., "in" and "out" dependencies) because no central node collects the necessary information; Instead, all the involved nodes collectively detect the deadlock. This requires that $(\downarrow T_detect_x)_{i_j} \geq max(T_in_{i_j}, T_out_{i_j})$ for all the nodes i_j on the path. The event in the system where a path is detected is such that the ends of the completed intervals on the nodes on the path lie in its past cone, and all information received until then must support the fact that those nodes indeed form a path. Thus, for distributed detection, the characterization of deadlock needs to be extended by examining the timestamps at the ends of intervals on the nodes.

DEFINITION 5. A *path* $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ at T_detect_x is a "confirmed" path if for all nodes i_j on the path, $(\downarrow T_detect_x)_{i_j} \geq max(T_in_{i_j}, T_out_{i_j})$, i.e., a completed interval at each node on the confirmed path can be observed at T_detect_x.

Not only do nodes on a confirmed path know that the nodes before them on their segment are blocked, but also the first nodes on segments (i.e., nodes which have OI intervals) in the confirmed path know at the end of their interval that all the nodes in the previous segment are blocked. A deadlock detected at T_detect_x is a closed "confirmed" path at T_detect_x.

A node i_j can ascertain the following about the system state: i_{j-1} is waiting on i_j at $T_out_{i_{j-1}}$ and i_j is waiting on i_{j+1}

at $T_out_{i_j}$, only at or after the end of the interval. A node i_j can confirm involvement in a deadlock only if a node is waiting on it and it is waiting on some node. Observe that the following conditions hold at the end of (completed) OI intervals on the nodes with OI intervals on the confirmed path:

(1) $T_out_{i_j}[i_j] \geq T_out_{i_{j-1}}[i_j]$;

(2) $i_{j+1} \in out(i_j)$ during the interval;

(3) $T_out_{i_{j-1}}[i_{j-1}] = T_in_{i_j}[i_{j-1}]$.

Constraint (1) follows because i_j remains blocked from $T_out_{i_j}$ until $(\downarrow T_detect_x)_{i_j}$ and because it does not send a message with a timestamp greater than $T_out_{i_j}[i_j]$ in this period, as per Axiom 1. Constraint (2) follows because i_j remains blocked from $T_out_{i_j}$ until $(\downarrow T_detect_x)_{i_j}$. Constraint (3) follows because at the end of its interval, i_j does not perceive i_{j-1} as having moved its local clock beyond $T_out_{i_{j-1}}[i_{j-1}]$ which is exactly the time when i_{j-1} got blocked on i_j on the current dependency, as per Axiom 1.

Due to the distributed nature of the dependency information, i_{j-1}'s dependency on i_j can belong to $\downarrow T_{i_j}$ only when i_j receives a valid request from i_{j-1} at $T_in_{i_j}$. A node with an IO interval on a path can know that all nodes before it on *its* segment are blocked when it blocks at $T_out_{i_j}$ because dependencies are formed serially and this information is valid along $\downarrow T_out_{i_j}$ (the information actually becomes known sooner at $T_in_{i_j}$). A node i_j with a completed OI interval (i.e., the first node on a segment) can know at $T_in_{i_j}$ that all the nodes in the *previous* segment are blocked. These conclusions can be made at $T_in_{i_j}$ because the blocked events of nodes on the previous segment belong to $\downarrow T_in_{i_j}$.

Semi-Distributed Algorithm: The above scheme suggests a semi-distributed detection algorithm in which completed OI intervals are reported to a central node. When a node completes an OI interval (this will be the first node in a segment) at event $T_in_{i_j}$, it reports all the events on the preceding segment (which now belong to the past of $T_in_{i_j}$) to a central node rather than each node in the preceding segment reporting itself to the central node (as done in the scheme in Section 3.2.5). The outward dependency information of the nodes on a segment flows with the request when each of the nodes blocks, until it reaches the first node on the next segment. Hence, if the average length of a segment is s, and a closed path is of length d, the algorithm needs d/s message transmissions.

Distributed Algorithm: In a (fully) distributed deadlock detection algorithm, the more frequently a node broadcasts messages, the sooner other nodes learn of its latest state. A distributed deadlock detection algorithm can disseminate its state information by adapting a wide range of policies. At one extreme, when a node's blocked status changes, the changed status is broadcast to all other nodes. Each node in this broadcast algorithm behaves like the central node in the centralized scheme in Section 3.2.5. An example of this algorithm is the Isloor–Marsland algorithm [15]. At the other extreme, when a path is extended/formed, only the minimum number of messages to satisfy the requirements of deadlock detection are sent. Broadcast algorithms have high overheads, and so we focus on algorithms in which information is propagated in a directed manner.

All the dependency information needed to evaluate consecutive blocking and the global consistency of blocking at the observation events which was identified in Lemmas 1 and 2, respectively, needs to be propagated. This information was enumerated in the centralized detection scheme of Section 3.2.5. The information T_{i_j} of each i_j can be combined as it propagates to evaluate Lemma 2 on-line.

Examples

Several probe-based algorithms, e.g., [7, 33, 35], and path-pushing algorithms, e.g., [26], are based on the directed information propagation scheme outlined above. In probe-based algorithms, node i_j initiates a probe when it blocks (an outward dependency is formed). The probe is stored at node i_{j+1} if i_{j+1} is active. When i_{j+1} blocks on node i_{j+2}, i_{j+1} forwards a copy of i_j's probe to i_{j+2}. This is exactly when i_{j+1} has an IO interval. If i_{j+1} was blocked when it received i_j's probe (i_{j+1} has an OI interval), then i_j's probe has to be explicitly forwarded to i_{j+2} because it is not going to send any more computation messages. In this case, i_{j+1} is the first node on its segment.

Status of a deadlock as seen by i_j can change whenever a message arrives or an incident dependency is formed or broken. A node i_h may learn of an event at node i_j when (1) i_h receives a message from i_j, or (2) a message sent from i_j to i_k is received at i_k, after which a message is sent by i_k to i_h and is received at i_h. This requires intelligent interpretation of the timestamp of each received message in order to always observe the system on the surface of the past cone; imposing FIFO behavior on the communication channels does not solve the problem.

Deadlock detection messages should carry a vector timestamp to correctly evaluate Lemma 2 on-line. In Sections 5 and 6, we use the developed formalism to demonstrate that algorithms [7, 14, 35] are incorrect. Now on the discussion is restricted to a nonglobal observer.

4. DEADLOCK RESOLUTION

A deadlock resolution event represents the withdrawal of request(s) by a node on a closed path. Let the event of resolving a deadlock detected at T_detect_x be $T_resolve_k$. (Note that in general, the resolver and the detector of a deadlock may be different.) The correctness of a detected deadlock is examined by observing the relation among T_detect_x, $T_resolve_k$ and $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ in the timing diagram shown in Fig. 5. Let $abs(max_{(\forall i_j)}(T_out_{i_j}))$ be denoted by t_f, the global time at which deadlock forms.

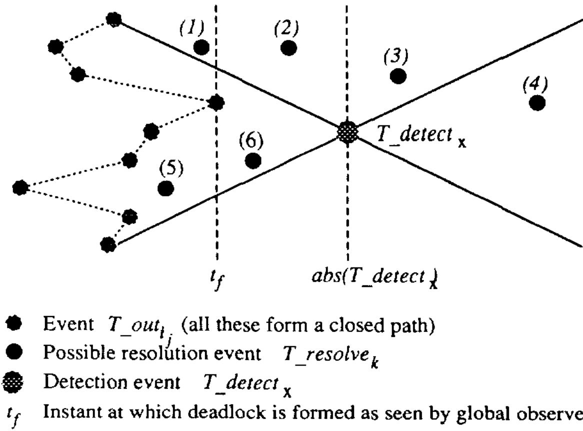


FIG. 5. Relationship between deadlock detection and resolution events.

Clearly, $\forall i_j$, $abs(T_detect_x) \geq abs(T_out_{i_j})$. There are six cases depending on the relationship between T_detect_x and $T_resolve_k$. These are denoted as (1)–(6) below and in Fig. 5.

(1) $abs(T_resolve_k) < t_f$ and ($T_resolve_k$ *co* T_detect_x). "Deadlock" is broken before it is formed at global time t_f. However, $T_resolve_k$ is not observable at T_detect_x because $T_resolve_k$ is concurrent with T_detect_x.

(2) $t_f < abs(T_resolve_k) < abs(T_detect_x)$ and ($T_resolve_k$ *co* T_detect_x). Deadlock is detected in global time after it has formed. However, in global time, the deadlock ceases to exist before it is detected. $T_resolve_k$ occurred concurrently with T_detect_x and cannot be observed at T_detect_x because it happens after $\downarrow T_detect_x$. This is an inherent limitation due to the lack of a global observer. Note that the first two cases are indistinguishable to the observer.

(3) $t_f < abs(T_detect_x) < abs(T_resolve_k)$ and ($T_resolve_k$ *co* T_detect_x). Deadlock exists at the global time when it is detected. Though it is resolved after it forms and after it is detected in global time, the resolution event occurs concurrently with the detection event and the resolution is not a consequence of the detection, i.e., $T_resolve_k$ does not lie in the future cone of T_detect_x. This case is indistinguishable from the previous two cases to the observer.

(4) $T_detect_x < T_resolve_k$. The resolution at $T_resolve_k$ occurs with knowledge about T_detect_x and lies in the future cone of T_detect_x.

(5) $abs(T_resolve_k) < t_f$ and ($T_resolve_k < T_detect_x$). "Deadlock" is broken before it is formed at global time t_f and resolution event $T_resolve_k$ is observable at T_detect_x. Nevertheless, the observer sees the corresponding closed path at the surface of the past cone $\downarrow T_detect_x$ because of an algorithmic error. Clearly, event T_detect_x falsely declares a deadlock.

(6) $t_f < abs(T_resolve_k) < abs(T_detect_x)$ and ($T_resolve_k < T_detect_x$). Deadlock is broken in global time after it formed and $T_resolve_k$ is observable at T_detect_x. Nevertheless, the observer sees the corresponding closed path at the surface of the past cone $\downarrow T_detect_x$ because of an algorithmic error. $T_resolve_k$ occurred before T_detect_x and can be observed at T_detect_x because it happens before $\downarrow T_detect_x$. Clearly, event T_detect_x falsely declares a deadlock.

5. FALSE DEADLOCKS

We now use the developed framework to define false deadlocks. Roesler and Burkhard [28] have defined two types of false deadlocks, viz., pseudo and phantom, in terms of physical global time. However, these definitions are rather naive and use physical global time which is impossible to realize. In this section, we define false deadlocks precisely under the state-theoretic framework developed here. It turns out that there are three types of false deadlocks, namely, shadow, phantom, and pseudo.

5.1. Shadow Deadlocks

DEFINITION 6. A *shadow* deadlock is a deadlock that is observed along $\downarrow T_detect_x$, but is resolved concurrently with the detection event T_detect_x. (Event T_detect_x detects a shadow of the deadlock that has been concurrently resolved.)

A shadow deadlock is observed due to the inherent limitation that at T_detect_x, an observer x can observe only what occurred up to $\downarrow T_detect_x$. The information along the $\downarrow T_detect_x$ indicates a deadlock but the deadlock resolution event cannot be observed because it is concurrent with T_detect_x. Nothing can be done to avoid shadow deadlocks (except in a "restricted model"; see Section 6) because they occur due to inherent limitations. No node involved in a shadow deadlock can determine the instant at which the deadlock is resolved because a concurrent resolution of the deadlock could be taking place. Shadow deadlocks are represented by cases (1), (2), and (3) in the previous section.

5.2. Phantom Deadlocks

DEFINITION 7. A *phantom* deadlock is a deadlock that is declared at T_detect_x because a closed path is observed along $\downarrow T_detect_x$, but the corresponding deadlock has been resolved in $Past(T_detect_x)$.

A phantom deadlock is a deadlock which existed in the system for a global observer or for a nonglobal observer (at some possibly nonexistent event), but is not present in the system along T_detect_x. Phantom deadlocks occur due to algorithmic errors. The main algorithmic errors are as follows: (i) information about outdated dependencies is not cleaned properly, (ii) outdated information is propagated, and (iii) temporal information which can capture causal relationships is not included in the deadlock detection process. Phantom deadlocks are represented by cases (5) and (6) in the previous section.

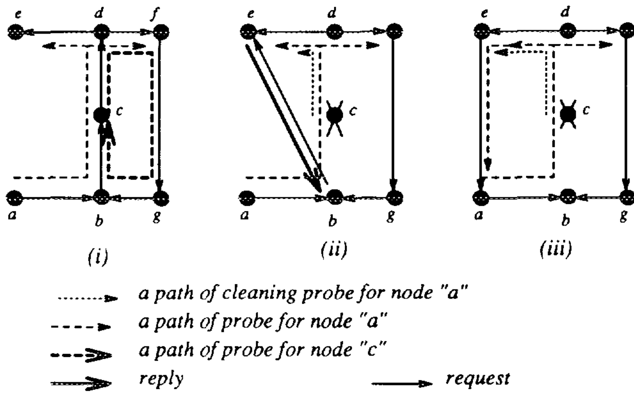


FIG. 6. An example of a pseudo deadlock in the Vossen and Wang algorithm [35].

5.3. Pseudo Deadlocks

DEFINITION 8. A *pseudo* deadlock is a deadlock that is declared at T_detect_x but the corresponding closed path $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ such that $T_out_{i_j} < T_detect_x$, never existed for any observation event.

Various segments and paths of a pseudo deadlock never existed simultaneously for a global or a nonglobal observer. The information along $\downarrow T_detect_x$ indicates no deadlock, but incorrect deductions are made by the algorithm. Pseudo deadlocks are reported due to algorithmic errors. The primary error is that an algorithm fails to include temporal information in the reasoning and wrongly concludes that a set of segments/paths, which existed at different times, constitute a closed path.

The algorithms by Vossen and Wang [35], Obermarck [26], Sinha and Natarajan [33], Choudhary *et al.* [7], Ho and Ramamoorthy [14] and Roesler and Burkhard [28] are some of the algorithms that declare pseudo deadlocks. The two-phase algorithm of [14] reports pseudo deadlocks because temporally unrelated blocking events that do not satisfy condition (C2) $\Rightarrow$ ((C3) $\wedge$ (C4)) are used to declare deadlocks. We next give a detailed example to show how the Vossen and Wang algorithm [35] (which is also a representative of the algorithms [7, 33]) detects pseudo deadlocks because it does not satisfy (C5′).

An Example

In the Vossen and Wang algorithm [35], a node circulates probe messages along the edges of the WFG and detects a deadlock when it receives its own probe. It then initiates cleaning probes which are propagated along the edges of the WFG to clean the probes that it had propagated in the WFG and then it aborts. The example is shown through three scenarios in Fig. 6.

Scenario (i): The probe initiated by node "c" returns to node "c" which detects deadlock. Node "c" sends out cleaning probes for all nodes (including node "a") whose probes it had propagated (i.e., forwarded).

Scenario (ii): Node "c" removes adjacent WFG edges and aborts. Node "b" becomes unblocked, blocks on node "e" which promptly replies to "b" and unblocks "b."

Scenario (iii): A WFG edge from node "e" to node "a" forms. Node "a" 's probe arrives back at "a" and node "a" declares deadlock "a-b-c-d-e." Node that the cleaning probe for node "a" is still chasing the probe—it had not reached node "e" when "e" blocked on "a."

Node "a" declares a pseudo deadlock because the deadlock could not have been observed by a global or a nonglobal observer. Note "e" 's past cone contained the fact that node "c" had aborted, yet it did not represent this information correctly. When "e" blocked, it propagated the probe for "a" without taking into account this information. Thus, node "a" got back an inaccurate state of the WFG and violated (C5′). This is a limitation of this algorithm because it cannot detect causal ordering of messages as probes don't carry timestamps.

Note that had the interaction between "b" and "e" not occurred in scenario (ii), then the deadlock detected by "a" would be a "shadow" deadlock. This is because "e" 's past cone would not contain the information that "c" had aborted.

In scenario (ii), assume that the interaction between "b" and "e" does not occur. Instead, the following occur: "b" replies to "a," "b" sends a request to "h" which is not shown (assume that two-phase locking [1] is not followed by "b"), "h" receives "b" 's request, "h" sends a request to "a," and "a" receives "h" 's request. Then in scenario (iii), when "a" receives its own probe from "e," it declares a phantom deadlock "a-b-c-d-e." This is a phantom deadlock because the closed path did exist for any omniscient observer but was resolved in "a" 's past cone. "a" could not recognize the resolution because it did not use vector clocks.

6. DEADLOCKS IN A RESTRICTED MODEL

A restricted model with each of the following features is identified:

- Single-request model.
- Nodes cannot abort spontaneously.
- A single resolution for a detected deadlock.

Since a node can make a single request at a time, deadlocks are isolated, i.e., a node can be part of at most one deadlock. Also, deadlock resolution cannot occur concurrently with detection of the deadlock because resolution occurs only as a result of detection. Since a node cannot spontaneously withdraw its request, deadlocks are stable, i.e., a deadlock persists until explicit resolution of it is performed. A very interesting consequence of all this is that the surface of the past cone $\downarrow T_detect_x$ becomes a vertical line $abs(T_detect_x)$.

In the restricted model, deadlock resolution can occur only as a result of detection due to stability of blocking

(i.e., $T_detect_x < T_resolve_k$), therefore, $abs(T_detect_x) < abs(T_resolve_k)$. Consequently, cases (1)–(3) in Fig. 5 can be avoided in this model. (They should be avoided because they lead to unnecessary resolutions.) Shadow deadlocks should never occur. When a deadlock is detected, it is guaranteed to exist in global time. (*Consecutive blocking* of nodes implies that $\langle T_out_{i_1}, T_out_{i_2}, \ldots, T_out_{i_n}\rangle$ is a *consistent cut*. So check **(3)** in the centralized detection scheme (Section 3.2.5) can be deleted.)

LEMMA 3. *In the restricted model, a deadlock which is detected at T_detect_x exists at $abs(T_detect_x)$.*

Proof. Follows because (i) each member of $\langle T_{i_1}, T_{i_2}, \ldots, T_{i_n}\rangle$ is blocked at a time in $Past(T_detect_x)$ and therefore, at a time less than $abs(T_detect_x)$; (ii) there is stability of blocking. ■

Note that stability of deadlocks is not guaranteed in the AND and the generalized request models because requests can be satisfied in the abnormal mode (case (ii) of Axiom 2). Specifically, stability of deadlocks is not guaranteed whenever a system permits the following: (i) a node can be a part of multiple closed paths (consequently, a closed path may get resolved due to a resolution of another closed path) or (ii) a node can spontaneously withdraw its request.

Several deadlock detection algorithms for the restricted model are incorrect. The algorithm by Vossen and Wang [35] works incorrectly even in this model. Sinha and Natarajan's algorithm [33] and Choudhary *et al.*'s algorithm [7] which are designed for the single request model also detect false deadlocks [20, 29]. These algorithms have the problem of undetected deadlocks, too. We next look at the false deadlock problem in [7].

An Example

Choudhary *et al.*'s algorithm [7] is probe-based and detects deadlocks by circulating probes in the WFG. Since "probes" and "cleaning probes" do not carry timestamp information, it is not possible to carry the timestamp of $T_resolve_k$. To avoid a situation where node k is detected as a part of the deadlock by x and $T_resolve_k < T_detect_x$, the algorithm uses a "freezing scheme" wherein the information of the resolution event is spread throughout the WFG (the closed path in this model), using "cleaning" probes, before a resolution is actually carried out. In doing so, the algorithm goes wrong and reintroduces information about the node which is about to be resolved into the system. This propagates without check and causes pseudo deadlocks to be detected. The algorithm is designed for a database system where Tr_i represents transaction i and DM_j represents the data manager for data j.

For example, in Fig. 7 suppose Tr_3 is the victim chosen to resolve the deadlock. Consider the following sequence of actions: (i) Tr_3 sends a "clean" message to DM_4 and waits for it to return before aborting. (ii) In the meanwhile, it receives Tr_5's probe from DM_5 and forwards the probe to DM_4. (iii) Tr_3 now receives its own "clean" message and aborts. (iv) Tr_2, Tr_1, and Tr_4 successively become active (after Tr_3 releases DM_3). (v) Tr_4 starts waiting transitively on Tr_5 through Tr_6 (not shown in Fig. 7). (vi) Tr_4 receives the Tr_5's probe from DM_4 and forwards it toward Tr_5 via Tr_6. (vii) The data manager where Tr_5 is the holder (not shown in Fig. 7) receives Tr_5's probe and detects a false deadlock (Tr_5, Tr_3, Tr_4, Tr_6).

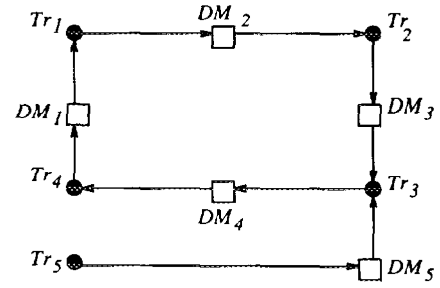


FIG. 7. An example of Choudhary *et al.*'s algorithm [7].

Note that Tr_4 becomes active only after Tr_3 aborts and Tr_2 and Tr_1 become active. The knowledge that Tr_3 has aborted lies in the past cone of Tr_4. Thus, Tr_4 could have discarded Tr_5's probe which was forwarded to it by Tr_3 (because Tr_3 has aborted since then).

The error of not viewing the system along the surface of the past cone appeared in the Vossen–Wang algorithm [35], too. An additional error in the algorithm [7] that is specific to the restricted model, is that in step (ii), Tr_3 reintroduces information that it is blocked and undoes the actions of the cleaning probe.

7. CORRECTNESS CRITERIA

The classical criteria for the correctness of distributed deadlock detection algorithms are as follows:

1. Progress (No undetected deadlocks): A distributed deadlock detection algorithm must detect all deadlocks in finite time.

2. Safety (No false deadlocks): A distributed deadlock detection algorithm must not report false deadlocks.

These criteria were directly taken from shared memory systems where the global system state is instantaneously and consistently available. These correctness criteria must be adapted for distributed deadlock detection to account for the fact that no node in a distributed system can have an instantaneous view of the system state. Each node deduces the status of all other nodes from all the knowledge available to it. The latest consistent knowledge that a node x can have about the system state at an instant $abs(T_detect_x)$ is represented by the surface of the past cone of the observation event T_detect_x. Therefore, all assertions about deadlock can be made only along the unique observable consistent cut $\downarrow T_detect_x$. Consequently, the correctness criteria for a distributed deadlock detection algorithm become as follows:

1′. Progress (No undetected deadlocks): A distributed deadlock detection algorithm must report all deadlocks (closed paths as per Theorem 1) which it can observe along $\downarrow T_detect_x$ for any observable event T_detect_x.

2′. Safety (No false deadlocks): A distributed deadlock detection algorithm should report a deadlock at T_detect_x only if it observes the corresponding closed path (as per Theorem 1) along $\downarrow T_detect_x$.

Correctness of distributed deadlock detection depends to a large extent upon observing the surface of the past cone $\downarrow T_detect_x$ consistently. However, not only should the surface of the cone be observed and inferred correctly, but also a distributed deadlock detection algorithm should take appropriate actions (such as promptly propagating information about wait-for dependencies and promptly cleaning or invalidating information about outdated dependencies caused by a resolution) to ensure that the surface of the cone represents the system state accurately. Therefore, besides the progress and safety conditions, we need an additional correctness condition for a distributed deadlock detection algorithm to ensure consistency of the information along the surface of the past cone. This correctness condition is given below:

3. Consistency of Information (surface of the past cone should be correct): The surface of the past cone $\downarrow T_detect_x$ of any observation event T_detect_x at node x must correctly reflect the complete system history contained in the past cone of the event.

If multiple nodes detect the same deadlock, only one of the nodes should be permitted to resolve the deadlock.

Note that many distributed deadlock detection algorithms are incorrect because they do not satisfy the third correctness condition. They do not take appropriate actions to ensure that the surface of the cone represents the system state accurately. Some algorithms do not propagate wait-for dependencies correctly; consequently, an existing deadlock may not be noticeable at the surface of the past cone, resulting in failure to detect existing deadlocks. Example of such algorithms are algorithms of Menasce and Muntz [25], Sinha and Natarajan [33], and Choudhary *et al.* [7]. Many algorithms do not clean/invalidate out-dated wait-for dependencies (such situations are created by a deadlock resolution or a spontaneous abort) correctly or do not collect the surface of the cone correctly; consequently, a deadlock cycle appears at the surface of the past cone even though there may not be a deadlock, resulting in detection of false deadlocks. Examples of the algorithms that do not clean wait-for dependencies in some cases are Sinha and Natarajan [33] and Choudhary *et al.* [7]. Examples of the algorithms that do not view the surface of the cone correctly are Gray [13] and Ho and Ramamoorthy [14]. These two algorithms do not satisfy condition **"2′. Safety"** and a deadlock is observed even though the corresponding closed path never existed.

To ensure that information about a deadlock appears on the surface of the past cone in finite time, all necessary deadlock detection activities for a distributed algorithm should have been initiated before or at the end of the *interval* at each node. A distributed deadlock detection algorithm must circulate the states of the nodes so as to bring all the blocking events within some node's view in a finite number of steps.

8. SUMMARY AND DISCUSSION

Detection of distributed deadlocks is difficult because no node in a distributed system has an instantaneous view of the system state. The definition of a deadlock and correctness criteria for a distributed deadlock detection algorithm must be redefined for distributed systems to take this into account.

This paper has presented a characterization of distributed deadlocks which explains the process of formation of distributed deadlocks in a state-theoretic framework. We have analyzed the formation of distributed deadlocks from the basic building blocks of deadlocks. We have examined the interaction between distributed deadlock detection and resolution which provides a deeper insight into the properties of distributed deadlocks. The characterization has resulted in a new set of correctness criteria for distributed deadlock detection algorithms.

Characterization of distributed deadlocks for a nonglobal observer is more complex than it is for a global observer because a nonglobal observer does not have an instantaneous view of the system state due to the inherent system limitations—lack of a global memory and a common physical clock. We have characterized distributed deadlocks from the basic building blocks, namely, segments and paths, and have discussed how a node (nonglobal observer) can observe them in terms of local variables and timestamps, in distributed systems. For an observation event T_detect_x, a nonglobal observer can base its decision on the information available only along $\downarrow T_detect_x$, the unique observable consistent cut at T_detect_x which represents the (nonvertical) surface of the past cone. This is equivalent to taking a global snapshot [3] along $\downarrow T_detect_x$, and assembling it at T_detect_x to check for deadlocks. What is effectively observed in this snapshot is the virtual global state that would have been reached if each node i_j in the snapshot initiated no further action after $(\downarrow T_detect_x)_{i_j}$.

An analysis of deadlock detection and resolution reveals that resolution may contribute to detection of false deadlocks. Deadlocks which are observed along the unique observable consistent cut at the observation event, but which are resolved concurrently with the observation event may be treated as "existing" deadlocks. We have defined three types of false deadlocks, viz., shadow, phantom, and pseudo deadlocks, in the developed framework.

The effect of a global observer can be achieved in a restricted system model that was identified. Here, stabil-

ity of blocking guarantees that a deadlock observed by a nonglobal observer along the surface of a past cone persists until the absolute time of the observation instant.

Based on our state-theoretic analysis of distributed deadlocks, we have defined a new set of correctness criteria for a distributed deadlock detection algorithm, which consists of the following three parts: (1) Every deadlock observed along the unique observable consistent cut $\downarrow T_detect_x$ must be reported. (2) All assertions about a deadlock at an event T_detect_x at node x must be made only along $\downarrow T_detect_x$. (3) A distributed deadlock detection algorithm should take appropriate actions (such as prompt propagation of information about wait-for dependencies and prompt cleaning of information about out-dated dependencies caused by a resolution) to insure that the surface of the cone represents the system state accurately. These three criteria essentially require that system state information should be correctly observed as well as correctly disseminated.

Future work can involve the characterization of deadlocks in the OR and generalized request models along the lines of the AND request model dealt with in this paper. Though the concept of *segments* and *paths* presented here holds for these request models also, there is a different semantics involved in these models.

ACKNOWLEDGMENTS

We thank the referees for their very useful comments.

REFERENCES

1. Bernstein, P. A., and Goodman, N. Concurrency control in distributed database systems, *ACM Computing Surveys* **13,** 2 (June 1981), pp. 185–221.
2. Bracha, G., and Toueg, S. Distributed deadlock detection, *Distribut. Comput.* **2** (1987), 127–138.
3. Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Systems* **3,** 1 (1985), 63–75.
4. Chandy, K. M., Misra, J., and Haas, L. M. Distributed deadlock detection. *ACM Trans. Comput. Systems* **1,** 2 (1983), 144–156.
5. Charron-Bost, B. Combinatorics and geometry of consistent cuts: Application to concurrency theory, *Proceedings of the 3rd International Workshop on Distributed Algorithms, 1989.* Lecture Notes in Computer Science, Springer-Verlag, New York/Berlin, 1989, vol. 392, pp. 45–56.
6. Charron-Bost, B. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* (July 1991) 11–16.
7. Choudhary, A. L., *et al.* A modified priority-based probe algorithm for distributed deadlock detection and resolution. *IEEE Trans. Software Eng.* (Jan. 1989), 10–17.
8. Coffman, E. G., Elphick, M., and Shoshani, A. System deadlocks. *ACM Comput. Surveys* (June 1971), 66–78.
9. Datta, A., and Ghosh, S. Synthesis of a class of deadlock-free Petri-nets, *J. Assoc. Comput. Mach.* (July 1984), 486–506.
10. Fidge, C. A. Timestamps in message-passing systems that preserve partial ordering. *Austral. Comput. Sci. Comm.* **10,** 1 (Feb. 1988), 56–66.
11. Fidge, C. A. Logical time in distributed computing systems. *IEEE Comput.* (Aug. 1991), 28–33.
12. Gligor, V. G., and Shattuck, S. H. On deadlock detection in distributed systems. *IEEE Trans. Software Eng.* **6,** 5 (1980), 435–440.
13. Gray, J. Notes on database operating systems. In *Operating Systems: An Advanced Course,* Springer-Verlag, New York, 1978, pp. 393–481.
14. Ho, G. S., Ramamoorthy, C. V. Protocols for deadlock detection in distributed database systems. *IEEE Trans. Software Eng.* **SE-8,** 6 (Nov. 1982), 554–557.
15. Isloor, S. S., and Marsland, T. A. An effective on-line deadlock detection technique for distributed database management systems. *Proc. COMPSAC 78,* (Nov. 1978), 283–288.
16. Knapp, E., Deadlock detection in distributed databases. *ACM Comput. Surveys* **19,** 4 (Dec. 1987), 303–328.
17. Kshemkalyani, A. D., and Singhal, M. Correct two-phase and one-phase deadlock detection algorithms for distributed systems. *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing, 1990,* pp. 126–129.
18. Kshemkalyani, A. D., and Singhal, M. Characterization of distributed deadlocks. Technical Report OSU-CISRC-6/90-TR15, The Ohio State University, June 1990.
19. Kshemkalyani, A. D. Characterization and correctness of distributed deadlock detection and resolution. Ph.D. dissertation, The Ohio State University, Aug. 1991.
20. Kshemkalyani, A. D., and Singhal, M. Invariant-based verification of a distributed deadlock detection algorithm. *IEEE Trans. Software Eng.* **SE-17,** 8 (Aug. 1991), 789–799.
21. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM* **21,** 7 (July 1978), 558–565.
22. Lamport, L. The mutual exclusion problem: Part 1: A theory of interprocess communication. *J. Assoc. Comput. Mach.* **33,** 2 (Apr. 1986), 313–326.
23. Mattern, F. *Virtual Time and Global States of Distributed Systems.* Parallel and Distributed Algorithms, North-Holland, Amsterdam, 1989, pp. 215–226.
24. Meldal, S., Sankar, S., and Vera, J. Exploiting locality in maintaining potential causality. *10th ACM Symposium on Principles of Distributed Computing, 1991,* pp. 231–240.
25. Menasce, D. A., Muntz, R. R. Locking and deadlock detection in distributed databases. *IEEE Trans. Software Eng.* **5,** 3 (1979), 195–202.
26. Obermarck, R. Distributed deadlock detection algorithm. *ACM Trans. Database Systems* (June 1982), 187–210.
27. Panengaden, P., and Taylor, K. Concurrent common knowledge: A new definition of agreement for asynchronous systems. *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing, 1988,* pp. 197–209.
28. Roesler, M., and Burkhard, W. A. Resolution of deadlocks in object-oriented distributed systems. *IEEE Trans. Comput.* **38,** 8 (1989), 1212–1224.
29. Sanders, B., and Heuberger, P. A. Distributed deadlock detection and resolution with probes. *Proceedings of the 3rd International Workshop on Distributed Algorithms, 1989.* Lecture Notes in Computer Science, Springer-Verlag, New York/Berlin, 1989, Vol. 392, pp. 207–218.
30. Shih, C-S. Distributed deadlock detection and its application to real-time systems. COINS Technical Report 92-60, University of Mass., Sept. 1992.
31. Singhal, M. Deadlock detection in distributed systems, *Computer* (Nov. 1989), 37–48.
32. Singhal, M., and Kshemkalyani, A. D. Efficient implementation of vector timestamps. *Inform. Process. Lett.* **43** (Aug. 1992), 47–52.
33. Sinha, M. K., and Natarajan, N. A priority-based distributed deadlock detection algorithm. *IEEE Trans. Software Eng.* (Jan. 1985), 67–80.

34. Tay, Y. C., and Loke, W. T. A theory for deadlocks. CS-TR-344-91, Princeton University, Aug. 1991.
35. Vossen, G., and Wang, S. S. Correct and efficient deadlock detection and resolution in distributed database systems. *Proceedings of the 5th International Conference on Data Engineering, 1989*, pp. 287–294.

AJAY D. KSHEMKALYANI received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987, and the M.S. and Ph.D. degrees in computer and information science from the Ohio State University, Columbus, in 1988 and 1991, respectively. He is currently an advisor in the Networking Systems Architecture Division in IBM at Research Triangle Park, North Carolina. His current research interests include distributed systems, operating systems, computer architecture, and databases. He is a member of the ACM and the IEEE Computer Society.

MUKESH SINGHAL is currently an associate professor of computer and information science at The Ohio State University, Columbus. He received the Bachelor of Engineering degree in electronics and communication engineering with high distinction from University of Roorkee, Roorkee, India, in 1980 and the Ph.D. degree in computer science from University of Maryland, College Park, in May 1986. His current research interests include distributed systems, operating systems, databases, and performance modeling. He has coauthored a book titled *Advanced Concepts in Operating Systems: Distributed, Multiprocessor, and Database Operating Systems* (McGraw–Hill, New York, 1994).

Received April 23, 1992; revised December 9, 1992; accepted January 27, 1993