Distributed Deadlock Detection

K. MANI CHANDY and JAYADEV MISRA
University of Texas
and
LAURA M. HAAS
IBM

Distributed deadlock models are presented for resource and communication deadlocks. Simple distributed algorithms for detection of these deadlocks are given. We show that all true deadlocks are detected and that no false deadlocks are reported. In our algorithms, no process maintains global information; all messages have an identical short length. The algorithms can be applied in distributed database and other message communication systems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.1 [**Operating Systems**]: Process Management—*deadlocks; synchronization*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*

General Terms: Algorithms

Additional Key Words and Phrases: Distributed deadlock detection, message communication systems, resource deadlock, communication deadlock

1. INTRODUCTION

In a system of processes which communicate only with a single central agent, deadlock can be detected easily because the central agent has complete information about every process. Deadlock detection is more difficult in systems where there is no such central agent and processes may communicate directly with one another. If we could assume that message communication is instantaneous, or if we could place certain restrictions on message delays, deadlock detection would become simpler. However, the only realistic general assumption we can make is that message delays are arbitrary but finite. In this paper, we present deadlock detection algorithms for networks of processes in which there is no single central agent and in which message delays are arbitrary but finite. The only assumption we make is that messages sent by process A to process B are received by B in the order in which they were sent by A.

This work has been supported by the Air Force Office of Scientific Research under grant AFOSR 81-0205 and by the University of Texas under a grant from the University Research Institute.

Authors' addresses: K. M. Chandy and J. Misra, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. M. Haas, IBM Research, 5600 Cottle Road, San Jose, CA 95193.

We consider two deadlock models in message communication systems: resource and communication deadlocks. Deadlock detection algorithms are given for both models. Most deadlock models in distributed databases are resource deadlock models [3, 4, 5, 8, 10, 11, 12, 14]; in these models, deadlock arises because processes may wait permanently for resources held by each other. The communication deadlock model is both more abstract and more general; it is applicable to any message communication system.

In the resource model, a process which requests resources must wait until it acquires *all* the requested resources before it can proceed with its computation. For example, a process may request resources *a*, *b*, and *c*; the process can proceed only after receiving all three resources—*a*, *b*, and *c*. The communication model is applicable to arbitrary resource requests involving the logical processes AND or OR. For instance, a process may require resources *a and* either *b or c*; if it receives *a*, it continues to wait for either *b or c*; if it receives *b* first, it must cancel its request for *c* and continue to wait for *a*. In general in the communication model, upon receiving any one resource the process sends cancellation messages if necessary and waits for a new set of resources. Although the deadlock detection algorithm given for the communication model can be applied to the resource model, the algorithm given in this paper for the resource model is simpler.

Many algorithms in the literature are incorrect in that they either fail to report some genuine deadlocks or report deadlocks where none exist. We show that our algorithms for both models detect all genuine deadlocks and report no false ones.

Dijkstra and Scholten presented an algorithm to detect termination in diffusing computations [2]. Diffusing computation is a model of distributed computations in which the computation is started by a special process, the *initiator*, which sends one or more messages. Processes other than the initiator can send messages only after receiving a message. Each process is ready to receive messages from all other processes at all times. Thus the computation terminates only when every process is idle, waiting for every other process. Our communication model is intended to support implementations of languages such as CSP [7], and therefore we must allow (1) a process to wait selectively for messages from *some* (not necessarily all) other processes, and (2) any process to send a message without having received a message. As a consequence, we must detect deadlock in our model when any *subset* of processes wait for each other, whereas in the Dijkstra and Scholten model, termination is detected only when *all* processes are waiting for all others. An algorithm for termination detection of diffusing computations in communicating sequential processes appears in [13].

Resource and communication deadlock models are introduced in Section 2. An algorithm for the resource model is given in Section 3 and one for the communication model is given in Section 4. Implementation issues are discussed in Section 5.

2. A MODEL OF DISTRIBUTED COMPUTATION

A network consists of a set of processes which communicate with one another exclusively by messages. We adopt the message communication protocol of Dijkstra and Scholten [2]: any message sent by one process to another is received correctly after an arbitrary but finite delay, and message transmissions obey the first-in-first-out rule, that is, messages sent by any process P_i to any other process P_j are received by P_j in the sequence in which they were sent by P_i. These requirements can be met by using sequence numbers and time-outs [9] and by having every process poll periodically for input messages.

A process P_i can assert only that any message it sends to process P_j will be received *eventually.* However, process P_i cannot assert that P_j has actually received the message unless it receives some form of acknowledgement from P_j. In our model, a process never waits to send a message. In CSP [7], by contrast, a process P_i can send a message to a process P_j only when P_j is willing to receive the message. The CSP protocol is implemented in our model by having the message sender first send the message and then wait for an acknowledgment; the message receiver sends an acknowledgment upon receiving the message. Thus, the sender can proceed with its computation only after the receiver has received the message.

At any given time, a process is in one of two states: idle or executing. Only processes that are in the executing state can send messages. An executing process may change its state (to idle) at any time. An idle process may change its state (to executing) only after its requests have been granted; the conditions for this state change are different for resource and communication models and are described in the following sections.

2.1 Resource Model

We study the resource deadlock problem as it arises in distributed databases (DDBs). A DDB consists of *resources, controllers,* and *processes.* Associated with each controller is a set of resources which it manages and a set of constituent processes. A process can only request resources from its own controller, but this controller may have to communicate with other controllers in order to reserve the particular resource. A process cannot execute unless it acquires *all* the resources for which it is waiting. A set of processes is said to be deadlocked when no process in the set can execute because each process requires a resource held by some other process in the set. Below is a more formal description derived from [12].

A DDB is implemented by N *computers* $S_1, \ldots, S_N$. A local operating system or *controller* C_j at each computer S_j schedules processes, manages resources, and carries out communication. There are M *transactions* $T_1, \ldots, T_M$ running on the DDB. A transaction is implemented by a collection of *processes* with at most one process per computer. Each process is labeled with a tuple P_{ij} where T_i is the identity of the transaction that the process belongs to and S_j is the computer on which the process runs.

At some stages in a transaction's computation it may need to *lock* resources (such as files). When a process P_{ij} needs a resource, it sends a request to its controller C_j. If C_j manages the resource, and if the particular resource is available, it may accede to the request immediately; otherwise, the process has to wait to acquire the requested resource. If the requested resource is managed by some other controller C_m, then C_j transmits the request to process P_{im} via controller C_m; now process P_{im} requests the resource from its controller C_m. When P_{im} acquires the requested resource from C_m, it sends a message to P_{ij} (via C_m and

C_j) stating that the requested resource has been acquired. P_{ij} may proceed with its computation only after it has received positive replies to *all* of its requests for resources. When processes in a transaction T_i no longer need a resource managed by controller C_m, they communicate with process P_{im} which is responsible for releasing the resource to C_m. Messages sent by any controller C_i to another C_j, arrive sequentially and in finite time.

A process cannot proceed with its computation until it acquires *every* resource that it requested. A set of processes is said to be *idle* when it is waiting to acquire a resource; it is said to be *executing* when it is not idle. Thus, if a process never acquires a requested resource, it is permanently idle. We assume that if a single transaction runs *by itself* in the DDB, it will terminate in finite time and eventually release all resources. When two or more transactions run in parallel, deadlock may arise.

For notational simplicity we assign a single identifying subscript (rather than a double subscript) to a process; P_i denotes the *i*th process. We refer to a process's controller and transaction explicitly (rather than by using subscripts). The use of a single subscript rather than a double subscript does not alter the problem at all; we merely renumber all the processes in the system with 1, 2, 3, ... A process P_j is said to be *dependent* on another process P_k if there exists a sequence (*seq*) of processes P_j, $P_{i(1)}$, . . . , $P_{i(m)}$, P_k, where each process in *seq* is idle and each process (except the first) in *seq* holds a resource for which the previous process in *seq* is waiting; P_j is defined to be *locally dependent* on P_k if all the processes in *seq* belong to the same controller. If P_j is dependent on P_k then P_j must remain idle at least as long as P_k remains idle. P_j is deadlocked if it is dependent on itself or on a process that is dependent on itself. In either case, deadlock exists only if there is a cycle of idle processes each dependent on the next process in the cycle. The goal of resource deadlock detection algorithms is to declare that deadlock exists *if and only if* such cycles exist.

2.2 Communication Model

The communication model is an abstract description of a network of processes which communicate via messages. There are no explicit controllers (or resources) in this model; controllers must be implemented by processes; requests for resource allocation, cancellation, and release must be implemented by messages.

Associated with every idle process is a set of processes called its *dependent set.* An idle process starts executing upon receiving a message from *any* process in its dependent set; otherwise, it does not change state or its dependent set. A process is *terminated* if it is idle and its associated dependent set is empty. For the moment, we assume that processes do not terminate, fail, or abort; these issues are considered in Section 5.

Intuitively, a nonempty set S of processes is deadlocked if all processes in S are permanently idle. A process is permanently idle if it never receives a message from any process in its dependent set.

It is not possible to detect permanent idleness in the following situation. Process A is waiting for a message from process B; process B is currently executing and will send a message to process A only upon completion of a loop; process A appears to be permanently idle if process B's loop computation is nonterminating. Detection of permanent idleness of this type amounts to solving the halting problem and hence is undecidable. We must therefore assume in this situation that A is not permanently idle since B *may* send it a message some time in the future. Therefore we adopt the following operational definition of deadlock. a nonempty set of processes S is *deadlocked* [15] if and only if

1. All processes in S are idle;
2. The dependent set of every process in S is a subset of S; and
3. There are no messages in transit between processes in S.

A process is *deadlocked* if it belongs to some deadlocked set.

A nonempty set S of processes satisfying the above three conditions must remain idle permanently because (1) an idle process P_i in S can start executing only after receiving a message from some process P_j in its dependent set, (2) every process P_j in P_i's dependent set is also in S and cannot send a message while remaining in the idle state, and (3) there are no messages in transit from P_j to P_i, which implies that P_i will never receive a message from any process in its dependent set.

2.3 A Comparison of Resource and Communication Deadlocks

There are several differences between the resource model and the communication model. One critical difference is that in the communication model, a process can know the identity of those processes from which it must receive a message before it can continue. If process A needs to receive a message from process B, then A can know that it is *waiting for B*. Thus, the processes have the necessary information to perform deadlock detection if they act collectively. In the resource model the dependence of one transaction on actions of other transactions is not directly known. All that is known is whether a transaction is waiting for a given resource or whether a transaction holds a given resource. A controller at each site keeps track of its resources and only the controllers can deduce that one transaction is waiting for another. Thus the agent of deadlock detection in the two environments is not the same.

The second major difference is that in a resource allocation model a process cannot proceed with execution until it receives *all* the resources for which it is waiting. In CSP and similar communication models, a process cannot proceed with its execution until it can communicate with *at least one* of the processes for which it is waiting. For instance, a process in CSP executing a guarded command may wait to receive from several processes; a guard succeeds and execution continues when a message is received from *any one* of these processes. The difference between the resource model and the communication model is between *waiting for all resources* and *waiting for any one message*; this difference results in very different algorithms for the two models.

In graph-theoretic terms, deadlock arises in the resource model when there is a *cycle* of (idle) dependent processes, whereas in the communication model there must be a *knot*[1] of (idle) waiting processes.

The communication model is more general than the resource model. In particular, the resource model can be simulated as a communication model. Further-

[1] A vertex i of a directed graph is a knot if all vertices that can be reached from i can also reach i.

more, the communication model can handle the case where a process waits for a logical combination of resources, as in resource *a and* resource *b* or resource *c*.

3. DEADLOCK DETECTION IN THE RESOURCE MODEL

A great deal of effort has gone into developing a distributed algorithm for detecting resource deadlocks in distributed databases (DDBs) [2, 3, 6]. In a recent paper, Gligor and Shattuck [3] state that "renewed interest in distributed systems has resulted in the publication of at least ten protocols for deadlock detection. However, few of these protocols are correct and fewer appear to be practical." Below, we present a solution [1] to this problem.

In order to determine whether an idle process is deadlocked, its controller initiates a *probe computation.* In a probe computation, controllers send messages called *probes* to one another. Probes are concerned exclusively with deadlock detection and are distinct from resource requests and replies. Probe computations may be initiated for several processes, and the same process may have several probe computations initiated for it in sequence.

A probe is a triple (i, j, k) denoting that it belongs to a probe computation initiated by P_i, and that this probe is being sent from process P_j on one controller to process P_k on another. The intuitive meaning of a probe(i, j, k) is as follows. P_j sends probe(i, j, k) to P_k when the following conditions exist: P_j is idle, P_j is waiting for (i.e., waiting to acquire resources from) P_k, and P_j has determined that P_i is dependent on P_j. This probe may be discarded or accepted by P_k; the probe is accepted by P_k if and only if P_k is idle, P_k did not know that P_i was dependent on it, and P_k can now deduce that P_i is dependent on it. It follows that if P_i accepts a probe(i, j, i), for any j, then P_i is deadlocked.

3.1 Algorithm

The controller maintains a Boolean array *dependent*$_k$ for each constituent process P_k, where dependent$_k(i)$ is *true* only if P_k's controller knows that P_i is dependent on P_k. If dependent$_i(i)$ is *true*, then P_i is dependent on itself and hence is deadlocked. Initially, dependent$_i(j)$ is set *false* for all i and j. The detailed algorithm for the probe computation is given below.

For initiation of probe computation by a controller for a constituent idle process P_i:

```
if $P_i$ is locally dependent on itself
  {i.e., $P_i$ belongs to a deadlocked set of processes,
   all on the same controller}
then declare deadlock
else for all $P_a$, $P_b$ such that
          (i) $P_i$ is locally dependent on $P_a$, and
         (ii) $P_a$ is waiting for $P_b$, and
        (iii) $P_a$, $P_b$ are on different controllers,
    send probe$(i, a, b)$.
```

For a controller on receiving a probe(i, j, k):

```
if
   (i) $P_k$ is idle, and
  (ii) dependent$_k(i)$ = false, and
 (iii) $P_k$ has not replied (positively) to all requests of $P_j$,
then begin
        dependent$_k(i)$ = true;
        if $k = i$
        then declare that $P_i$ is deadlocked
        else for all $P_a$, $P_b$ such that
                    (i) $P_k$ is locally dependent on $P_a$, and
                    (ii) $P_a$ is waiting for $P_b$, and
                    (iii) $P_a$ and $P_b$ are on different controllers,
              send probe($i$, $a$, $b$).
    end
```

For a controller when a constituent process P_k becomes executing:

set dependent$_k(i)$ = **false** for all i

The proof that a process is actually deadlocked when it is declared to be so follows from the fact (which may be shown by induction) that dependent$_k(i)$ is *true* only if P_i is dependent on P_k and P_k is idle. Conversely, we can prove that P_i will be declared to be deadlocked if a probe computation for P_i begins when there exists a cycle of processes P_i, $P_{j(1)}, \ldots, P_{j(m)}$, P_i, where each process in the sequence is dependent on the next. The proof follows from the inductive hypothesis: dependent$_{j(k)}(i)$ will be set to true and $P_{j(k)}$ will send a probe to the next process in the sequence for $1 \leq k \leq K$, for all K. Detailed proofs are given in [1].

If desired, when a controller detects that one of its constituent processes is deadlocked, it can inform (via their controllers) all processes waiting for the deadlocked process that they too are deadlocked.

4. DEADLOCK DETECTION IN THE COMMUNICATION MODEL

We now describe a deadlock detection algorithm for the communication model that will allow an idle process to determine whether it is deadlocked.

An idle process can determine whether it is deadlocked by initiating a *query computation* when it enters the idle state. The query computation is distinct from the underlying computation for which deadlock is being detected. Processes may exchange messages for the query computation even in the idle state. This is because the state is idle only in reference to the underlying computation. The process which initiates a query computation is called the *initiator* of that query computation. Several processes may initiate query computations and the same process may initiate query computations several times.

The messages in a query computation are of the form query(i, m, j, k) and reply(i, m, j, k), denoting that these messages belong to the mth query computation initiated by process P_i and are being sent from P_j to P_k. P_i, m, P_j, and P_k are called the initiator, sequence number, sender, and receiver, respectively. There will be at most one message of the form query(i, m, j, k); there will be at most one reply message of the form reply(i, m, k, j) to the query message query (i, m, j, k).

The query computations have the following properties.

1. If process P_i is deadlocked when it initiates its mth query computation, then it will receive reply(i, m, j, i) corresponding to every query(i, m, i, j) that it sent. (See Theorem 1 below.)

2. If the initiator, P_i, has received reply(i, m, j, i) corresponding to every query(i, m, i, j) that it sent, then it is deadlocked. (See Theorem 2 below.)

In our algorithm each process P_k maintains four arrays of local variables. The indices of the array range over all processes in the network. The local variables for P_k $k = 1, 2, 3, \ldots$ are described below.

Definition 4.1 The variable latest(i) is the largest sequence number m in any query(i, m, j, k) sent or received by P_k. Initially, latest(i) = 0, for all i.

Definition 4.2 The variable engager(i), where $i \neq k$, is the identity, say j, of the process which caused latest(i) to be set to its current value m by sending P_k the message query(i, m, j, k). The initial value is arbitrary.

Definition 4.3 The variable num(i) is the total number of messages of the form query(i, m, k, j) sent by P_k, minus the total number of messages of the form reply(i, m, j, k) received by P_k, where m = latest(i) and j is arbitrary. Note that num(i) = 0 means that P_k has received replies to all queries of the form (i, m, k, r) that P_k sent, where m = latest(i).

Definition 4.4 The variable wait(i) is *true* if and only if P_k has been idle continuously since latest(i) was last updated. Initially wait(i) is false, for all i.

4.1 Algorithm

An idle process initiates a query computation by sending queries to processes in its dependent set. The basic idea is that an idle process on receiving a query should propagate the query to its dependent set if it has not done so already. Thus, if there is a sequence of permanently idle processes $P_i, \ldots, P_j$, such that each process in the sequence (except the first) is in the dependent set of the previous process in the sequence, a query initiated by P_i will be propagated to P_j.

For the remainder of this discussion we consider the action taken by a process P_k on receiving a query or reply with initiator i, sequence number m, and sender j. Local variables latest(i), engager(i), num(i), and wait(i) refer to the variables of process P_k. If $m <$ latest(i), then P_k discards the message because (by Definition 4.1) P_i must have initiated the query computation with sequence number latest(i) after it initiated the mth query computation; hence P_i could not have been deadlocked when it initiated the mth query computation. If wait(i) is false when P_k receives the query/reply, and if m = latest(i), then (by Definition 4.4) P_k has been in the executing state since it first participated in the mth computation initiated by P_i. In this case, we can assert (by Theorem 1 below) that P_i was not deadlocked when it initiated its mth computation; hence P_k, discards the message. Thus, P_k discards all messages except those in which $m >$ latest(i) or those received when wait(i) is true and m = latest(i).

If $m >$ latest(i), then by Definitions 4.1, 4.2, and 4.4, P_k must set latest(i) to m, engager(i) to j (where P_j is the sender), and wait(i) to true. If P_k receives a reply in which m = latest(i) and wait(i) is true, then (by Definition 4.3), P_k must decrement num(i) by 1.

When P_k receives a query in which $m >$ latest(i), it propagates the query to all processes in its dependent set, sets num(i) to the number of processes in the dependent set (by Definition 4.3), and updates other local variables as discussed earlier. When P_k initiates a query computation it does so by acting as though it had just received a query in which $m >$ latest(k).

Next, we derive the conditions under which P_k sends replies. If wait(i) is true when P_k receives a query in which m = latest(i), it replies to the query immediately. If wait(i) is false when P_k receives a query in which m = latest(i), or wait(i) is arbitrary and $m <$ latest(i), then P_k discards the query and *never* replies to this query. The interesting question is when should P_k reply to a query in which $m >$ latest(i)? By Definitions 4.1 and 4.2, such a query must have been sent by engager(i). The answer to this question is derived by extending the conditions in Dijkstra and Scholten [2]; P_k replies to engager(i) only if wait(i) is true and num(i) has been reduced to zero (i.e., P_k has received replies to all queries with initiator P_i and sequence number m that it sent, and P_k has been continuously idle since it first participated in this query computation). This scheme for replying to engagers is necessary for Theorem 2 (given below): if the initiator receives replies to all the queries it sends to its dependent set, then the initiator is deadlocked. The detailed description of the algorithm is given below.

For an idle process P_i to initiate a query computation:

```
begin
    latest($i$) := latest($i$) + 1; wait($i$) = true;
    send query($i$, latest($i$), $i$, $j$) to all processes $P_j$ in
    $P_i$'s dependent set $S$; num($i$) := number of elements in $S$
end
```

For an executing process P_k: On becoming executing, set wait(i) = *false*, for all i. Discard all queries and replies received while in executing state.

For an idle process P_k, upon receiving query(i, m, j, k):

```
if $m >$ latest($i$)
then begin
        latest($i$) := $m$; engager($i$) := $j$; wait($i$) := true;
        for all processes $P_r$ in $P_k$'s dependent set $S$
        send query($i$, $m$, $k$, $r$);
        num($i$) := number of processes in $S$
     end
else if wait($i$) and $m$ = latest($i$)
     then send reply($i$, $m$, $k$, $j$) to $P_j$
```

Upon receiving reply(i, m, r, k):

```
if $m$ = latest($i$) and wait($i$)
then begin
        num($i$) := num($i$) − 1;
        if num($i$) = 0
        then if $i = k$
          then declare $P_k$ deadlocked
          else send reply($i$, $m$, $k$, $j$) to $P_j$
          where $j$ = engager($i$)
     end
```

4.2 Proofs

THEOREM 1. *If the initiator of a query computation is deadlocked when it initiates the computation, it will (eventually) declare itself deadlocked.*

PROOF. Consider a set S of processes, including the initiator, which is deadlocked at the instant at which the query computation is initiated. Processes in S

cannot change their dependent sets and there are no messages in transit between processes in S. Hence the problem is equivalent to that considered by Dijkstra and Scholten [2] where "queries" correspond to their "messages," and their proof applies here. □

In the following we restrict our attention to a single query computation, say the mth initiated by process P_i. Thus queries, replies, engagers, and so forth, refer to this specific computation.

LEMMA 1. *Suppose a process P_k sends a reply to its engager and subsequently becomes executing at some time, say t. Then there exists a process P_r in P_k's dependent set (where the dependent set is determined at the point at which P_k replies to its engager) which receives a query and subsequently becomes executing at some $t' < t$.*

PROOF. In order for P_k to reply to its engager, it must have received replies from all processes in its dependent set. In order for P_k to become executing, it must have received a message from some process in its dependent set, say P_r. Since P_k became executing after sending the reply to its engager, it must have received the message from P_r after the reply from P_r; hence, P_r must have sent the message after the reply. Therefore, the sequence of events must be as follows. P_r gets the query from P_k, replies to it, becomes executing, and sends the message to P_k causing it to become executing. □

THEOREM 2. *If the initiator of a query computation declares itself "deadlocked," then it belongs to a deadlocked set.*

PROOF. Let S be the set of processes including the initiator which received queries during this query computation. We will show that S is a deadlocked set. Every process replying to its engager in this computation must have received replies for all queries that it sent. From this fact, the following inductive hypothesis can be established. If the initiator declares itself deadlocked, a reply must have been received to the qth query in the computation, for $q = 1, 2, \ldots$. Therefore, every process in S replies to its engager in the computation. Hence, from Lemma 1, it follows that if process P_k in S becomes executing at time t after receiving a query, then some process P_j in S becomes executing at time t' after receiving a query, and $t' < t$. Using this inductively, it follows that no process in S can become executing after receiving a query, and hence S is a deadlocked set. □

THEOREM 3. *At least one process in every deadlocked set will report "deadlocked" if every process initiates a new query computation whenever it becomes idle.*

PROOF. From Theorem 1, the last process to become idle in a deadlocked set will report "deadlocked." □

As in the resource model, any process detecting deadlock can inform others that it is deadlocked.

4.3 Example

The resource model is so simple that we do not give an example for it. We do give an example for the communication model. A more elaborate example is found in [6]. In this example, we use subscripts to distinguish local variables belonging to different processes. For instance, engager$_i$$(j)$ is the value of engager(j) in process P_i.

Consider a network consisting of four processes. The network's initial state and a possible execution sequence is given below.

Initial State

Process 1: idle, waiting for process 2 or process 3
Process 2: idle, waiting for process 4
Process 3: idle, waiting for process 1 or process 4
Process 4: executing

Execution Sequence

Time	Action
1	Process 1 initiates its first query computation. Process 1 sends query(1, 1, 1, 2) and query(1, 1, 1, 3).
2	Process 2 receives query(1, 1, 1, 2). Engager$_2$(1) := 1. Process 2 sends query(1, 1, 2, 4).
3	Process 3 receives query(1, 1, 1, 3). Engager$_3$(1) := 1. Process 3 sends query(1, 1, 3, 1) and query(1, 1, 3, 4).
4	Process 4 sends a message to process 3.
5	Process 1 receives query(1, 1, 3, 1). Process 1 sends reply(1,1,1,3).
6	Process 4 changes state from executing to idle and waits for process 2. Note that processes 2 and 4 are now deadlocked.
7	Process 4 receives query(1, 1, 2, 4). Engager$_4$(1) := 2; latest$_4$(1) := 1; wait$_4$(1) := true. Process 4 sends query(1, 1, 4, 2); num$_4$(1) := 1.
8	Process 4 receives query(1, 1, 3, 4). Since m = latest$_4$(1) and wait$_4$(1) = true. Process 4 sends reply(1, 1, 4, 3).
9	Process 3 receives message from process 4 (sent at time 4). Process 3 becomes executing, sets wait$_3$(1) := false.
10	Process 4 initiates its first query computation. Process 4 sends query(4, 1, 4, 2); num$_4$(4) := 1; latest$_4$(4) := 1; wait$_4$(4) := true.
11	Process 2 receives query(1, 1, 4, 2) (sent at time 7). Process 2 sends reply(1, 1, 2, 4).
12	Process 3 receives reply(1, 1, 4, 3) (sent at time 8). Process 3 is executing. Wait$_3$(1) is false. Thus process 3 will never send reply(1, 1, 3, 1) and process 1 will not declare itself deadlocked.

13 Process 2 receives query(4, 1, 4, 2) (sent at time 10).
Engager$_2$(4) := 4*i*; wait$_2$(4) := true.
Process 2 sends query(4, 1, 2, 4); num$_2$(4) := 1.

14 Process 4 receives reply(1, 1, 2, 4).
Wait$_4$(1) is true and now num$_4$(1) is 0; engager$_4$(1) = 2.
Process 4 sends reply(1, 1, 4, 2).

15 Process 4 receives query(4, 1, 4, 2). Since latest$_4$(4) = 1, it sends reply(4, 1, 4, 2).

16 Process 2 receives reply(4, 1, 4, 2). Since wait$_2$(4) = true, it reduces num$_2$(4) from 1 to 0. Since engager$_2$(4) = 4, it sends reply(4, 1, 2, 4).

17 Process 4 receives reply(4, 1, 2, 4). Since wait$_4$(4) = true, it reduces num$_4$(4) from 1 to 0 and declares itself deadlocked.

5. NOTES ON THE ALGORITHMS

Two algorithms have been proposed for the detection of deadlocks. The first algorithm, applicable to resource deadlocks, is simpler, involving only one type of message (probes) for deadlock detection. The second algorithm is applicable to a more general class of problems and therefore involves two types of messages (query and reply). These algorithms are easy to implement since in either case, each message (probe, query, and reply) is of fixed length and requires a few simple computational steps.

These algorithms seem attractive for reasons of performance as well as correctness, since the overhead of deadlock detection computations and the message traffic associated with deadlock detection is generated *primarily when processes are idle* (i.e., they have nothing to do and nothing to send). Furthermore, executing processes need only discard the messages associated with a deadlock detection computation. Every single deadlock detection computation involves no more than e probes in resource models or e queries and replies in communication models, where e is the number of communicating process pairs in the network. In the worst case, where the network of N processes is fully connected, $e = N \times (N - 1)$. Normally, e, and hence the number of these messages, will be much less. For example, in the case where each of the N processes has a dependent set of size k or less, $e \leq k \times N$.

To reduce the number of deadlock detection computations which are initiated, a process may initiate one only if it has been idle continuously for some time T, where T is a performance parameter. If the process leaves the idle state before T, we have avoided initiating such a computation. Time-outs may be used in a similar manner for probe, query, and reply propagation. Note that since every process could initiate deadlock detection computations one or more times, proper choice of T is critical in reducing the number of these computations. Issues related to this are discussed in [5].

We can ensure that no process has a backlog of an unbounded number of queries or probes by requiring a process to receive acknowledgments to earlier queries or probes from a specific process before sending the next query or probe to that process.

Our algorithms require that all processes which are permanently idle (including terminated, failed, or aborted processes) reply to queries and propagate the probe.

If failure prevents such a reply/probe from being sent, the failure must be detected by other means and the reply/probe sent. Our algorithm further requires that the computation (for which deadlock is being detected) be correct. In particular, if the dependent set of a process is miscalculated, the deadlock detection algorithm may not function.

ACKNOWLEDGMENTS

We are particularly grateful to E. W. Dijkstra and C. S. Scholten for their encouragement and advice. In particular, the proof of Theorem 2 was suggested by Scholten. We were helped greatly by comments from G. Andrews, N. Francez, C. A. R. Hoare, Anita Jones, and F. Schneider. Special thanks to the referees for their comments.

REFERENCES

1. Chandy, K.M., and Misra, J. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing* (Ottawa, Canada, August 18-20, 1982), ACM, New York, 1982, pp. 157-164.
2. Dijkstra, E.W., and Scholten, C.S. Termination detection for diffusing computations. *Inf. Process. Lett. 11*, 1 (Aug. 1980), 1-4.
3. Gligor, V.D., and Shattuck, S.H. Deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-6*, 5 (Sept. 1980), 435-440.
4. Goldman, B. Deadlock detection in computer networks. Tech. Rep. MIT-LCS-TR185, Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1977.
5. Gray, J.N. Notes on database operating systems. In *Operating Systems: An Advanced Course*, vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York 1978, pp. 393-481.
6. Haas, L.M. Two approaches to deadlock in distributed systems. Ph.d. dissertation, Computer Science Dept., Univ. of Texas at Austin, July 1981.
7. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666-677.
8. Isloor, S.S., and Marsland, T.A. An effective 'on-line' deadlock detection technique for distributed database management systems. In *Proc. COMPSAC 1978*, IEEE, New York, pp. 283-288.
9. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558-565.
10. Lomet, D.B. Coping with deadlock in distributed systems. Res. Rep. RC 7460 (#32196), IBM, T. J. Watson Research Center, Yorktown Heights, N.Y., Dec. 1978.
11. Mahoud, S.A., and Riordon, J.S. Software controlled access to distributed databases. *INFOR 15*, 1 (Feb. 1977), 22-36.
12. Menasce, D., and Muntz, R. Locking and deadlock detection in distributed databases. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 195-202.
13. Misra, J., and Chandy, K.M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst. 4*, 1 (Jan. 1982), 37-43.
14. Obermarck, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June 1982), 187-208.
15. Chandy, K.M., and Misra, J. Deadlock absence proofs for networks of communicating processes. *Inf. Process. Lett. 9*, 4 (Nov. 1979), 185-189.

Received May 1981; revised August 1982; accepted November 1982