Correct Two-Phase and One-Phase Deadlock Detection Algorithms for Distributed Systems

Ajay D. Kshemkalyani and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210

Abstract

In this paper, we propose a correction to the two-phase deadlock detection algorithm [2], which was shown to be incorrect in [3]. We prove the correctness of the modified algorithm using a stable property detection technique that observes the system at an absolute time instant. We then use the notion of consistent cuts [8] and vector time [7] to give a simple one-phase deadlock detection algorithm that requires fewer messages and is faster than the one-phase algorithm in [2].

1 Introduction

Distributed systems are often prone to deadlocks [4, 9]. Therefore, detecting distributed deadlocks is an important problem. Several algorithms have been proposed for distributed deadlock detection, many of which have been shown to be incorrect because their reasoning is not based on a consistent wait-for graph (WFG) [4, 9]. Timestamps play a vital role in identifying different events in a distributed system to achieve reasoning about consistent global states [1, 7]. In this paper, we show how timestamps can be used for simple distributed deadlock detection algorithms. We propose a correction to the two-phase deadlock detection algorithm [2], which was shown to be incorrect in [3]. We prove the correctness of the modified algorithm using a stable property detection technique that observes the system at an absolute time instant. We extend the technique using the notion of virtual time [7] to observe the system along a consistent cut [8] to detect a stable or an unstable property. The virtual clocks proposed by Mattern [7] are used to give a simple one-phase deadlock detection algorithm that requires fewer messages and is faster than the one-phase algorithm in [2].

In Section 2, we give a correct two-phase algorithm. In Section 3, we present a one-phase algorithm. Section 4 contains the conclusions.

2 A Two-Phase Algorithm

In the distributed system model of [2], a process can initiate several transactions, where a transaction is defined as a sequence of request, lock and unlock operations. Each transaction in the system has a unique identification. Each site maintains a status table for the processes it initiates. The status table of a process keeps track of the resources the process has locked and resources the process is waiting for. The two-phase protocol operates as follows: (1) Periodically, some control site collects the status tables of each site (by broadcasting a request for the status tables and waiting until they arrive) and constructs a wait-for graph (WFG) of the system from the information received (Phase 1). (2) If there is a cycle in the WFG, the control site collects the (possibly changed) status tables of each site again (Phase 2) and forms a WFG using only those transactions reported in phase 1. If the WFG contains the same cycle, the control site declares the deadlock.

It was shown in [3] using a counter-example that the above protocol detects false deadlocks. There may exist multiple request/assignment edges involving the same transaction and resource, each due to a lock request issued at different times. Any two such edges exist over non-overlapping intervals of time. The above protocol cannot distinguish between two such edges. It wrongly declares deadlock when it detects that for each request (assignment) edge in a cycle in the WFG constructed after phase 1, there exists some request (assignment) edge involving the same resource and transaction in the WFG constructed after phase 2. However, a deadlock can be declared only if the same request (assignment) edges forming a cycle exist in the WFGs formed after phases 1 and 2.

2.1 Modified Two-Phase Algorithm

In the proposed modification to the two-phase algorithm [2], each edge as reported by a status table is identified by a triplet (T, R, t) which for a request edge means that transaction T made a request for resource R at time t and

TH0328-5/90/0000/0126/$01.00 © 1990 IEEE

Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:42:53 UTC from IEEE Xplore. Restrictions apply.

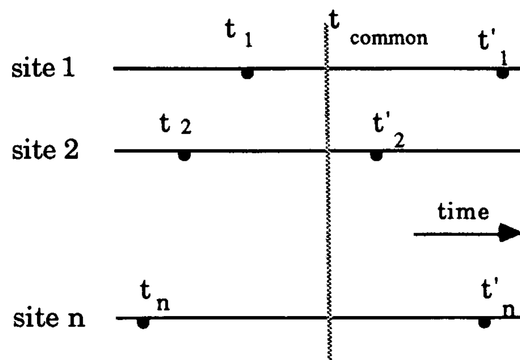

Figure 1: A timing diagram for the relationship among t_i, $t_i^{'}$ and t_{common}.

for an assignment edge means that transaction T received a grant reply for resource R at time t. The time t used above is the local clock value of the transaction and is operated by Lamport's rules [5]. Observe that each edge can be constructed and destroyed only once. The deadlock detection condition in phase 2 of the protocol is changed as follows: *If the WFG constructed after phase 2 contains a cycle composed of edges that are the same as those in the cycle detected after phase 1, then the control site detects a deadlock.*

Theorem 1 *The modified two-phase centralized deadlock detection algorithm does not report false deadlocks.*

Proof : In each phase, the control site does not achieve any coordination among the views of the status tables at various sites that it receives. Let the control site construct the WFG for phases 1 and 2 using the status tables of site i at instants t_i and $t_i^{'}$, respectively. (t_i, $t_i^{'}$ and all other times referred to in this proof are absolute global times. Though these times are not realizable, they are useful in constructing the proof.) Phase 2 is initiated after phase 1 ends. Hence, for all sites i and j, $t_i < t_j^{'}$. Since time is over a continuous domain, there exists a time t_{common} such that for all processes i, $t_i < t_{common} < t_i^{'}$. The relationship among t_i and $t_i^{'}$ on each site i, and t_{common} is depicted in Figure 1. When the control site detects a deadlock, it is guaranteed that exactly the same edges existing in the cycle detected after phase 1 exist in the cycle detected after phase 2. For each transaction at site i, the edges incident on it at $t_i^{'}$ existed at t_i and thus at t_{common}. Thus, a WFG cycle existed at t_{common}. Note that if all the edges in a cycle exist at the same (absolute) time, then there is a deadlock. So no false deadlock is reported. □

Our proof illustrates a technique that can be used for stable property detection in two-phase protocols. If an auxiliary property holds between the times that each site participates in phases 1 and 2 of the protocol, then there exists an instant in absolute time at which the auxiliary property holds. Developing a two-phase protocol involves identifying an auxiliary property such that (a) the auxiliary property holds at each site between the times that the site participates in phases 1 and 2 and therefore holds at an absolute instant t_{common} at all sites, and (b) if the auxiliary property holds at some absolute instant, then it implies the stable property the protocol seeks to detect (here a cycle).

An example of the use of this technique is the four-counter two-phase termination detection algorithm [6].

The auxiliary property used in the modified algorithm is that the same request/assignment edges forming a cycle exist in the WFGs constructed after phases 1 and 2. This auxiliary property meets conditions (a) and (b) listed above. Hence, detection of the auxiliary property implies that a deadlock exists. The algorithm in [2] uses the auxiliary property that for each request (assignment) edge in the WFG cycle constructed after phase 1, there exists some request (assignment) edge involving the same resource and transaction in the WFG constructed after phase 2. This auxiliary property does not meet condition (a). Hence, detection of the auxiliary property does not imply that a true deadlock exists.

3 One-Phase Algorithm

The two-phase algorithm above used a technique to identify a stable property in the system by reasoning about the system at an absolute time instant t_{common}. In general, observing a distributed system at an absolute time instant is not realizable; rather a global snapshot along a consistent cut provides a notion of virtual time that is intrinsic to a distributed system [1, 7, 8]. We give a one-phase algorithm for deadlock detection that views the system along the latest observable consistent cut; the technique used to view the system is useful for detecting stable and unstable properties.

In the system model, a process can make requests for exclusive access to resources and blocks when it makes the requests (the AND request model [4]). Each resource is managed by a resource manager. We will use the term node to refer to a process as well as a resource manager. In a system with n processes, the algorithm requires n messages to detect deadlocks. The algorithm has the following advantages over the one-phase algorithm presented in [2].

1. Message complexity is halved from $2n$ to n (although the size of the messages increases) because only pro-

Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:42:53 UTC from IEEE Xplore. Restrictions apply.

cesses (not resources) are required to report to the control site.

2. A deadlock can be detected faster because the control site does not have to wait until a dependency between a process and a resource is reported by both the proces and the resource, as is required by the algorithm in [2]. Rather, a deadlock can be detected as soon as all involved processes have reported their incident dependencies.

The algorithm uses an important property of vector clocks which are explained next [7].

3.1 Vectored Timestamps

Each node in the system maintains a vector clock whose time is the best approximation to the latest state in the system [7]. The logical time is defined to be a vector of length n, the number of processes. Note that a resource manager also has a clock but does not have an entry for its resource in the clock vector. The logical time at node i is T_i and the timestamp of a message msg is $msg.T$. ($T_i[j]$ and $msg.T[j]$, respectively denote j^{th} component of these time vectors.) The logical time at a node evolves as follows:

(a) When an internal event or a message send/receive occurs on process i, $T_i[i] := T_i[i] + 1$.

(b) When node i receives a message msg, then $\forall j$ do $T_i[j] := max(T_i[j], msg.T[j])$.

Thus, j^{th} component of time vector at a node reflects the highest value of the jth component of all timestamped messages it has received. Note that only $T_i[i]$ reflects the local activities at process i and $T_i[j]$ reflects what node i knows about the local timestamp (i.e., activities) of process j. Thus, time vector T_i reflects what node i knows of the latest state (local time) of all other processes. We identify events by referring to their (unique) timestamps.

An ordering relation "$<$" between timestamps is defined as follows: $T_i < T_j$ iff $(\forall k), T_i[k] \leq T_j[k]$. $T_i < T_j$ iff the event at T_i happened before [5] 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.

It is shown in [7] that events $T_1, T_2, \ldots, T_n$ lie on a consistent cut iff

$$sup(T_1, T_2, \ldots, T_n) = [T_1[1], T_2[2], \ldots, T_n[n]]$$

where $sup(T_1, T_2, \ldots, T_n) = T$ such that $(\forall i), (T[i] = max(T_1[i], \ldots, T_n[i]))$.

Remark 1 *Given events* $T_1, T_2, \ldots, T_n$, $T = sup(T_1, T_2, \ldots, T_n)$ *denotes the latest observable consistent cut on nodes* 1 *to* n *even if* $T_1, T_2, \ldots, T_n$ *do not lie on a consistent cut.*

3.2 The Algorithm

Each process maintains a process table which has entries (R, s) where R is the resource it is waiting for/holds, and s is its blocked status. s can be either w if the process is waiting for the resource or a if the process has been assigned the resource. A process i also maintains a local variable T_block_i which at any time is defined to have the value of the timestamp when i blocked on a request, if i is currently blocked.

The system operations are as follows:

1. When a process i wants to request resource R: i updates its component of its clock, sends a timestamped request to the resource manager of R, and enters (R, w) in its process table. T_block_i is assigned the current clock value.

2. When the resource manager for R receives a request from process i: it updates its clock.
(a) If R is free, it is assigned to i through a timestamped message and a lock is set on R.
(b) If R is not free, the request is placed in a queue.

3. When a process i receives a reply assigning it resource R: it updates its clock. The entry (R, w) in the process table is changed to (R, a) and T_block_i is assigned the value $\bar{0}$ if i is no longer blocked.

4. When a process i releases R: it updates its component of its clock, deletes the entry (R, a) from the process table, and sends a timestamped message to the resource manager of R to unlock the resource.

5. When the resource manager of R receives a message releasing the lock on R: it updates its clock and unlocks the resource. (It can now assign the resource to some process whose request is pending in the queue.)

The deadlock detection algorithm run periodically by a control site operates as follows:

1. The control site broadcasts a message to all processes i to send their timestamps, and if they are blocked, the variable T_block_i and the process table. The control site waits until all replies are received.

2. The control site constructs a wait-for-graph (WFG) as follows: For each process i that reports itself as blocked, it checks if $T_block_i[i] \geq T_j[i]$ for all values of $j \neq i$. If the above condition is true, then for entries (R, w) in i's proces table, the edge $i \longrightarrow R$ is added to the WFG and for entries (R, a) in i's process table, the edge $R \longrightarrow i$ is added to the WFG.

3. The control site declares a deadlock if there is a directed cycle in the WFG.

Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:42:53 UTC from IEEE Xplore. Restrictions apply.

Remark 2 *A blocked process i does not send out any message until it unblocks. Therefore, if process i reports itself as blocked and $T_block_i[i] < T_j[i]$ for some $j \neq i$, then i's information that it is blocked is out-dated and invalid with respect to the latest consistent cut T observable by the control site.*

The control site constructs a snapshot of the system along the latest observable consistent cut T using the timestamps of uncoordinated events, one on each process. It checks whether the blocked status reported by a process is reflected in T. If so, the dependency information reported by the process is consistent with T and is used in the WFG; if not, the information is out-dated and therefore discarded. This is the key idea in this algorithm.

Theorem 2 *A system is in a deadlock iff there is a directed cycle in the WFG constructed by the above algorithm.*

Proof:
1) *The necessary condition: If there is a deadlock, there is a directed cycle in the WFG constructed.* If there is a deadlock, no deadlocked process has sent out a message in which its component of the timestamp is greater than the local clock value at which it deadlocked. Therefore, in a WFG constructed, for no process i that deadlocked at T_block_i will there exist another process j such that $T_j[i] > T_block_i[i]$. If the system is deadlocked, all processes i in the deadlock must have reached $T_block_i[i]$ in some WFG that observes the system along the latest observable consistent cut T. When the WFG constructed uses information reported by each process i after T_block_i, it will contain a cycle because none of the (R, w) and (R, a) edges of processes in the deadlock are out-dated.
2) *The sufficient condition: If the WFG contains a directed cycle, there is a deadlock.* The central site observes the system along the latest observable consistent cut T. If process i reports itself as blocked at T_block_i and $T_block_i[i] < T_j[i], j \neq i$, then by Remark 2, the information reported by i is out-dated. The WFG is constructed using information reported by only those processes that are blocked along the latest observable consistent cut. Thus, no invalid edges are considered in the WFG. The result follows. □

4 Conclusions

We have given a correction to the two-phase deadlock detection algorithm [2] by using timestamps. We prove the algorithm correct by reasoning about the system state at an absolute time instant. Absolute time is overly restrictive; we can instead use virtual time [7] to observe the global state of the system along a consistent cut [8]. Using this, we gave a one-phase algorithm to detect deadlocks. The algorithm uses half the number of messages used by the algorithm in [2] and can detect deadlocks faster. The algorithm can easily be extended to more complex request models such as the P-out-of-Q request model [4] and can also factor in deadlock resolution because of which deadlock is not a stable property in the system [9].

References

[1] K.M. Chandy, L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, 63-75, 3(1), 1985.

[2] G. S. Ho, C. V. Ramamoorthy, *Protocols for Deadlock Detection in Distributed Database Systems,* IEEE Transactions Software Engineering, 554-557, SE-8(6), Nov. 1982.

[3] J. R. Jagannathan, R. Vasudevan, *Comments on "Protocols for Deadlock Detection in Distributed Database Systems"*, IEEE Transactions Software Engineering, 371, SE-9(3), May 1983.

[4] E. Knapp, *Deadlock Detection in Distributed Databases,* ACM Computing Surveys, 19(4), 303-328, Dec. 1987.

[5] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System,* CACM, 558-565, 21(7), July 1978.

[6] F. Mattern, *Algorithms for Distributed Termination Detection,* Distributed Computing, 2:161-175, 1987.

[7] F. Mattern, *Virtual Time and Global States of Distributed Systems,* Parallel and Distributed Algorithms, North-Holland, 215-226, 1989.

[8] P. Panengaden, K. Taylor, *Concurrent Common Knowledge: A New Definition Of Agreement for Asynchronous Systems,* Proceedings of the 5th ACM Symposium on Principles of Distributed Computing, 197-209, 1988.

[9] M. Singhal, *Deadlock Detection in Distributed Systems,* Computer, 37-48, Nov. 1989.

Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:42:53 UTC from IEEE Xplore. Restrictions apply.