DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

AJOY KUMAR DATTA * SUKUMAR GHOSH

Computer Science and Electrical Engineering Department
University of Nevada, Las Vegas
Las Vegas, NV 89154

Department of Computer Science
University of Iowa
Iowa City, IA 52242

Abstract

This paper deals with a method of detecting deadlocks in resource sharing for distributed systems. The algorithm is based on sending messages along the edges of the waitfor graph, and is built on a prioritized signalling mechanism which can be implemented on an underlying routing protocol. The proposed algorithm avoids the detection of false deadlocks, and is capable of detecting deadlocks involving a subset of processes in the system. The algorithm works well even when multiple nodes initiate the deadlock detection algorithm. A comparison of this algorithm with other existing distributed deadlock detection algorithms is also briefly presented.

1 INTRODUCTION

A distributed system consists of a collection of processes which communicate with each other to achieve a common goal. These processes may run concurrently on separate physical processors which are not connected to any global memory. Local states of the processes are maintained in memories local to the processors running the processes. In absence of a shared memory, processes communicate through messages. The communication is asynchronous, and a message may take an arbitrary but finite amount of time to move from one process to another. With improvements in microprocessor technology and the availability of viable computational models [6], distributed computation has attained substantial importance in the past few years.

In this universe of processes forming a distributed system, each component process has a set of local resources owned and managed by it. A process however uses resources which may or may not be local to it. A remote resource can be accessed by sending an explicit request to the owner of that resource. Such resources are often not shareable - exclusive access becomes necessary either due to hardware constraints, or due to software constraints like consistency and determinacy. When one process needs a resource currently being used by another process, it sends a request and waits for that resource to be released. It is assumed that every process is well-behaved in as much as once it acquires a resource, it releases it within a finite amount of time. It is fair to assume that no process has a prior knowledge of the future resource requirements of any process, including itself, in the total system.

In such a resource sharing environment, deadlock is a potential danger. When a set of processes enter into such a state that each process waits for some other process in this set to release a resource, all the processes are blocked indefinitely. In a shared memory environment, a number of deadlock detection algorithms are available [Holt 1972]. However, in a distributed environment, the added complexity of these algorithms is due to the unpredictable propagation delays, and the consequent nonavailability of the global state. With the exception of [1] most of the other known deadlock detection algorithms [9,11] first undergo a state collection process, and then detect possible cycles in the waitfor graph. Due to inconsistency in the collected global state, many of these algorithms are prone to false deadlocks [4]. The present algorithm utilizes some properties of the waitfor graph to minimize the state collection procedure, and embeds a signalling mechanism to overcome false deadlocks.

For survey articles on deadlock detection the readers may refer to [3,8,12].

2 BASIC CONCEPTS

2.1 The Waitfor Graph

Let the distributed system be composed of a set of n processes \{p_1, p_2, p_3, \ldots, p_n\} which are expected to share the resources belonging to them in such a way that deadlock does not occur. A waitfor graph [7] is a graph which represents which process is waiting for which other process for the purpose of acquiring a resource. A directed arc from some process \(p_i\) to another process \(p_j\) (Figure 1) would thus mean that \(p_i\) is using some resource which is also required by \(p_j\), and \(p_i\) can use it only after \(p_j\) releases that resource.

---

*This research was supported in part by the U. S. Army Research Office under grant DAAL03-87-G-0004.
Before embarking on further details, we make an important practical assumption which is as follows:

**Assumption 1:** Each process can request for only one resource at a time.

In case a process needs more than one resource, it can request for a second resource only after the request for the first resource has been honored. In a deadlock-free resource-sharing distributed system, the waitfor graph is a tree (or more precisely a forest), and under the assumption made above, the following property should hold for a waitfor graph:

**Lemma 1:** Every node in the waitfor graph can have at most one outgoing edge.

**Definition 1:** In the waitfor graph, a node with zero outgoing edges will be called a terminal node, and a node with zero incoming edges will be called an initial node.

In a tree of the waitfor graph, there should thus be only one terminal node (with zero outgoing edges), and it is this node which is eligible to make a request for a resource at the present moment.

Figure 1 shows a sample waitfor graph.

### 2.2 Building a signalling mechanism

When a process $p_i$ needs a resource owned by another process $p_j$, it sends a request to $p_j$, and sets up a provisional arc from $p_i$ to $p_j$ (Figure 2). If the resource is available, then the request is granted to $p_i$, and the provisional arc is removed. If on the other hand, the resource is not available, then a message is forwarded to the last process $p_k$ which requested for that resource, and the provisional arc is shifted from $p_i$ to $p_k$. The process $p_k$ eventually acknowledges this request which is communicated to $p_i$ by sending an ack (acknowledgement) signal via $p_j$, and the provisional arc from $p_i$ to $p_k$ is converted to a true arc from $p_i$ to $p_k$.

The communication of the request and the acknowledgement signals between a pair of processes is thus channelized through the owner of the concerned resource. Every process maintains a queue of the processes waiting for a resource, and this queue is maintained for every resource which it owns. In addition, every waiting process also knows the identity of the process whom it is waiting for. When a process completes using a resource, it sends a grant signal, and sets up that resource to the next waiting process by sending a release signal. This results in the removal of an arc in the waitfor graph. A release signal is also sent when a process is preempted due to any reason.

The pair of signals request, ack[request] constitutes an atomic action in as much as every process sending a request defers the decision about a subsequent request until it receives the corresponding acknowledgement signal, and every process receiving a request defers the decision about a subsequent request until it sends the corresponding acknowledgement signal. However, this oversimplified signalling mechanism itself is prone to deadlock. To break a possible deadlock in the signalling mechanism, we use the process numbers associated with the nodes.

a. Each process would henceforth append its own identity to the body of the request message. As in a priority interrupt scheme, requests with a higher process number, would get a higher priority over requests with a lower process number. A typical situation is illustrated in Figure 3. Here, process $p_1$ is trying to acquire a resource being used by $p_3$, and $p_3$ is trying to acquire a resource being used by $p_1$ with no other processes currently competing for these resources. However, since the process number contained in the request from $p_3$ is higher than the process number of $p_1$, $p_3$ is committed to accept and process the request first by sending an ack[request] to $p_3$, before receiving the acknowledgement of its own request.

In the present scheme, to detect a possible deadlock in the system, a process would send a find.deadlock signal to its successor node in the graph. If the successor node is a terminal node, it returns an ack[false] to the sender process down the edge of the waitfor graph, which implies that there is no deadlock. If however the successor node is not a terminal node, then it forwards the find.deadlock signal to its successor. An acknowledgement is sent to the predecessor node only after an acknowledgement is received from the successor node. An ack[true] would indicate the presence of a deadlock, whereas an ack[false] would indicate the absence of it.

While detecting the absence of deadlocks is fairly straightforward, detecting the presence of deadlocks poses a termination problem, since the waitfor graph becomes cyclic. The paper [2] provide a simple solution to overcome this problem.

**Lemma 2:** A node in the waitfor graph sends or forwards the find.deadlock signal to its successor node at most once. In case this node receives a second find.deadlock signal, it behaves as a terminal node and returns an ack[true] signal to the last sender of this signal.

The above approach overcomes the termination problem and helps detect deadlocks using a finite number of messages.

### 2.3 More about Signals

The set of signals introduced so far can be summarized as follows:

1. Request, ack[request]
2. find.deadlock, ack[true or false]
3. release, ack[release], grant.

These signals are supervisory signals and are distinct from the usual interprocess communication signals, which are an integral part of the underlying computation. Any undesirable interaction amongst these supervisory signals can be conveniently overcome by defining a priority structure in accordance with their numerical ordering in the list — where request has the lowest priority, and release & grant has the highest priority (Figure 4). To ensure the
absence of deadlock in the signalling mechanism itself, it is important to ensure that for every signal sent or forwarded by a node, the corresponding \texttt{ack[\_\_\_]} is eventually received. Since a signal with a higher priority can interrupt the atomic actions initiated by another signal with a lower priority, the indefinite blocking is avoided with signals of unequal priority levels. However, since the waitfor graph is a dynamic graph, an additional complication is possible which needs careful attention.

Consider that a node \texttt{p\_1} has received and forwarded a \texttt{find\_deadlock} signal to its successor \texttt{p\_2} and is waiting for the acknowledgement. Meanwhile due to some reason, the process \texttt{p\_2} has been aborted 1 (Figure 1), which leads to the removal of the arc from \texttt{p\_1} to \texttt{p\_2} in the waitfor graph. In the absence of this edge, the acknowledgement signal cannot return to \texttt{p\_1} leading to a deadlock in the signalling mechanism itself!

To overcome such problems, the following approach seems feasible:

\textbf{Definition 2}: A node in the waitfor graph will be called a \textit{pending node}, if it has initiated or forwarded a signal but has not yet received the corresponding acknowledgement.

\textbf{Lemma 3}: If a pending node which has sent or forwarded a \texttt{find\_deadlock} signal becomes a terminal node, then it should return an \texttt{ack[false]} to its predecessor. If a pending node becomes an initial node, then it should discard all the subsequent acknowledgements received by it.

With this modification, there cannot be a deadlock involving signals of unequal priorities. There is no apprehension of deadlock involving a number of \texttt{find\_deadlock} signals originating from different processes, since all such signals would flow in the same direction following the directed edges of the waitfor graph. Similar observations are valid for the release signals also. Thus, the signalling mechanism itself is free from deadlock.

3 THE ALGORITHM

3.1 Signal Handling

The life of a process in this universe of processes can be described as follows:

\begin{verbatim}
* [compute]
  [send request; receive ack[request] or grant ]
  [receive request; handle request; send ack[request]]
  [send release; receive ack[release]]
  [receive release; handle release; send ack[release]]
  [send find\_deadlock]; receive ack[true or false]
  [receive find\_deadlock]; process find\_deadlock;

send ack[true or false]
]

The algorithms for handling the signals request, release, grant and find\_deadlock are outlined below:

\textbf{var}
\texttt{p, succ : process;}
\texttt{r : resource;}
\texttt{q : array [1...r] of queue;}
\texttt{terminal : boolean;}
\texttt{pending : boolean;}
\texttt{free, owner : array [1...r] of boolean;}
\texttt{terminal is true only when the node is a terminal node; pending is true only when the node has sent a find\_deadlock signal and is waiting for ack[true] or ack[false]; initially, all the nodes are terminal nodes; q is empty for every resource; pending = false; succ = empty()}

\{receiving a request for a resource r\}
\texttt{p? request(r) →}
\texttt{[owner[r] →}
\texttt{[free[r] → [free[r] := false; p! grant(r)]]}
\texttt{[~ free[r] →}
\texttt{[tail.q[r]! request(r); tail.q[r]! ack[request(r)]; enqueue (p, q[r]); p! ack[request(r)]]}
\texttt{[~ owner(r) → p! ack[request(r)]]}

\{receiving an ack[request(r)]\}
\texttt{p? ack[request(r)] → [succ := p; terminal := false]}

\{receiving a release of a resource r\}
\texttt{p? release(r) →}
\texttt{[This is received by the owner of r only]}
\texttt{[dequeue(p, q[r]); p! ack[release(r)]; empty(r) → free[r] := true; ~ empty(r) → head.q[r]! grant(r); ]}

\{receiving a grant resource r\}
\texttt{p? grant(r) →}
\texttt{[succ := p; terminal := true; use resource r]}

\{receiving an ack[release(r)]\}
\texttt{p? ack[release(r)] → [no action]}
\end{verbatim}

\footnotesize
\begin{enumerate}
  \item The \texttt{grant} signal is of type \texttt{acknowledgement}
  \item Note that \texttt{release} has the highest priority for obvious reasons.
  \item The \texttt{find\_deadlock} signal is sent only by a process waiting for a resource.
\end{enumerate}

133
(receiving a find.deadlock signal)
p?find.deadlock →
  terminal → p! ack[false]  
  pending → [p! ack[true]; pending := false]
  pending → [succ? find.deadlock; pending := true]

(receiving an ack[true] or ack[false] signal)
initiating node →
  [succ? ack[true] → there is a deadlock]
  succ? ack[false] → there is no deadlock
other nodes →
  [succ? ack[true] → p! ack[true]]
  succ? ack[false] → p! ack[false]

3.2 Proof

The absence of deadlock in the signalling mechanism has already been established in Sections 2.2 and 2.3. This subsection only deals with a proof of the absence of false deadlocks.

Theorem 1: The proposed algorithm only detects true deadlocks.

Proof: False deadlocks are detected when the collected global state is an inconsistent one. In the proposed algorithm, in response to a find.deadlock signal, an ack[true] is received if:

- a node waiting for the ack[true] or ack[false] signal receives a second find.deadlock signal.

Since all the signals propagate along the edges of the waitFor graph, one can conclude that the initiator node is involved in a circular waiting. In this cycle of nodes, if a process is abruptly terminated before passing on the ack[true] signal to its predecessor node in the graph, then that predecessor node becomes a terminal node, and it returns an ack[false] instead of an ack[true] signal, indicating that the deadlock does not exist. Thus only true deadlocks are detected.

Note: If however, a process aborts itself after passing on the ack[true] signal to its predecessor node and before this ack[true] eventually reaches the initiator node, then one might foresee an apparent problem of a false deadlock being reported to the initiator node. However, in absence of a global clock, the terms before and after do not have any global significance — these are only defined by the happened-before relationship! This case therefore should not be considered as a case of a false deadlock being detected. This could possibly influence a future recovery action, which has not been dealt with in this paper.

4 CONCLUSION

Primitive state collection process can lead to the detection of false deadlocks, since the collected global state may be an inconsistent one. Such algorithms are therefore of little practical use in distributed deadlock detection. By propagating the probe signals along the edges of the waitFor graph, a process can derive consistent information about the termination state. The algorithms due to Dijkstra & Scholten (1980) and Chandy et al. (1982) belong to the latter category. The present algorithm is also based on the concept of chasing the edge of the waitFor graph, and has similarity with the work by Chandy et al. (1982). The emphasis of this algorithm however lies in the foundation of a strong signalling mechanism used in a supervisory capacity. Every signal is treated as an interrupt, and the prioritization of these signals (Figure 4) leads to a better coordination of the various messages which might originate from multiple initiator nodes. The proposed algorithm detects deadlock even when a subset of processes have entered into a deadlock. Careful thoughts were given to the assignment of priority levels to the different signals. For example, life would have been simpler if release were given the lowest level priority. However, it was given the highest priority because a last minute release might probably enable a node to send an ack[false] instead of an ack[true] thereby preventing a possible deadlock. However, a different way to optimize these signals is not ruled out.

5 REFERENCES


Figure 1: A sample waitfor graph.

Figure 3: Avoidance of deadlock in the signalling mechanism.

Figure 4: The priority structure amongst the signals.
Figure 2: The building of a waitfor graph.
(a) $p_4$ requests for a resource owned by $p_3$.
(b) This resource is currently being used by $p_7$, and $p_6$ is the last process in the queue for this resource.
(c) $p_6$ eventually sends an ack[request] to $p_4$. 