An $O(n)$ distributed deadlock resolution algorithm

Manuel Prieto, Jesús Villadangos, Federico Fariña, Alberto Córdoba
Dept. de Matemática e Informática
Universidad Pública de Navarra.
Campus Arrosadía, 31006 Pamplona, Spain.
{manuel.prieto, jesusv, fitxi, alberto.cordoba}@unavarra.es

Abstract

This paper shows a new distributed algorithm for deadlock detection and resolution under the single-resource request model that highly improves the complexity measurements of previous proposals. The algorithm has a communication cost of $2n-1$ messages and a latency of $n \cdot T$ for a deadlock cycle of $n$ processes, where $T$ is the inter-site communication delay. The algorithm achieves this improvement even verifying the strongest correctness criteria considered in previous works: it resolves all deadlocks in finite time and does not resolve false deadlocks.

Keywords: Distributed systems, Deadlock detection/resolution, Single-resource request model, Distributed algorithms, Complexity.

1 Introduction

A set of processes is deadlocked if every process in that set waits for resources held by other processes in that set [6]. The existence of deadlocks reduces the system throughput because deadlocked processes never terminate their execution and the resources they hold are not available to any other process [8, 13].

Usually, a deadlock is resolved by aborting a deadlocked process. When a process aborts, it cancels its pending request and releases the resources that it holds. The work done by the aborted process is wasted. Sometimes, aborted processes must be restarted in order to complete their work. Obviously, the abortion increases the response time of the process because it has to perform the work previously wasted again.

A correct deadlock detection/resolution algorithm must satisfy the following correctness criteria [8]:

Liveness Every deadlock is resolved in finite time.
Safety The algorithm does not resolve false deadlocks.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Message complexity</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chandy [1]</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Mitchell [11]</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Kshemkalyani [9]</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>González [4]</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Kim [7]</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Proposal</td>
<td>$2n-1 \sim O(n)$</td>
<td>$n \cdot T \sim O(n)$</td>
</tr>
</tbody>
</table>

Table 1. Deadlock resolution algorithms comparative table.

These correctness criteria are considered under the assumption that all the abortions are caused by the algorithm; that is, there are no spontaneous abortions.

In this paper we center our discussion on the single-resource request model (SR model). It is the simplest one, since the processes are restricted to request only one unit of a resource at each time. A deadlock in this model is characterized by a cycle in a graph representing the wait-for relations, so a deadlock is resolved by aborting a process in the deadlock cycle [8]. Because of its simplicity, this model has been widely used to model resource acquisition [14].

1.1 Previous works

Several distributed deadlock detection algorithms for the SR model have been proposed in previous works [1, 2, 11, 15, 3, 9, 4]. The most widely used method for distributed deadlock detection is edge-chasing. Originally proposed by Chandy et al. [1, 2], an edge-chasing algorithm detects a deadlock by propagating messages through the edges of the wait-for relations. A deadlock is detected when a message comes back to its initiator.

Since any blocked process or resource may initiate an instance of the algorithm and different instances do not collaborate (do not share information) in the graph exploration process, the message complexity of those algorithms is $O(n^2)$, where $n$ is the number of processes in the deadlock cycle (see table 1).
Another important issue is the correctness criteria. Algorithms that do not offer a mechanism for coordinating different instances of the algorithm can lead to the abortion of more than one process in order to resolve the same deadlock. Moreover, as the abortion of a process can avoid the formation of other deadlocks, an instance could detect such a non-formed deadlock.

1.2 Contributions of the paper

In this paper we show a deadlock resolution algorithm for the SR model that only needs \(2n - 1\) messages to resolve a deadlock of \(n\) processes. Thus, our algorithm can be considered an improvement over previous proposals because it has a lower communication cost while it keeps the lowest latency.

In addition, we show that the algorithm is safe (it never resolves false deadlocks) thanks to a novel coordination mechanism.

1.3 Overall organization

The rest of the paper is organized as follows. Section 2 describes the system model considered in the paper. Section 3 specifies the problem to solve. Section 4 explains the main characteristics of the proposed algorithm, shows its formal description and depicts its performance with an example. Section 5 offers a sketch of the correctness proof. Section 6 analyzes the complexity of the algorithm. Finally, conclusions and references end the paper.

2 System model

A distributed system can be viewed as a set of sites at which processes and resource managers are executed. Resource managers can be considered as processes, so we call all of them nodes. We consider a set of nodes \(N\). Each node has a system-wide unique identifier. There is a total order relation among node identifiers. Nodes communicate among themselves by message passing. We assume that there is a logical communications channel connecting each pair of nodes. Such communications channels are FIFO and reliable (messages are neither lost nor replicated). Message delays are arbitrary but finite.

In our system, the underlying computation follows the single-resource request model: a node gets blocked when it requests access to another node (a process waits for the grant message of a requested resource, while a resource manager waits for a release message to get again the resource control). In the SR model, a node waits for at most one other node.

Each site has a copy of the distributed deadlock resolution algorithm (DDRA). The manager informs the algorithm about the wait-for dependencies among the processes and the resources. The algorithm detects deadlocks and informs the manager about them in order to resolve those situations by aborting a process. It is assumed that the algorithm only aborts nodes representing processes. When a process is aborted, it withdraws its resource request, relinquishes all the resources it has acquired, and restores all the relinquished resources to their original state. There are no spontaneous abortions in the system; that is, no process aborts by itself.

In the following, we use an Input/Output Automaton [10], denoted \(S\), to formally describe the system (see figures 1 and 2). The state of \(S\) comprises: for each node \(i\), a set of node identifiers \((in_i)\) that represents the set of nodes which node \(i\) ‘thinks’ are waiting for it, a set of node identifiers \((out_i)\) that represents the set of nodes which node \(i\) is waiting for (note that as we model the SR this set is composed of at most by one element), and a status variable \((status_i)\) that represents whether node \(i\) is active, blocked or aborted; and for each pair of nodes \(i\) and \(j\), there is a FIFO queue \((ch(i, j))\) that represents the communications channel between node \(i\) and node \(j\). The steps of \(S\) are defined based on a precondition/effect schema. An action \(\pi\) is enabled from any state \(s\) satisfying its preconditions. The action can take the system from a state \(s\) to the new state \(s',\) if \(s'\) can be obtained by modifying \(s\) as indicated by the effects of the action.

We model the behavior of the system taking into account only the relevant information for the DDRA at each site: the changes in the local knowledge about the wait-for relations.

The execution of the \(\text{addOutArc}_i(j)\) action models that node \(i\) begins to wait for node \(j\) (i sends a \textit{request} or \textit{grant} message to node \(j\)) while \(\text{addInArc}_i(j)\) models that \(i\) finds out that \(j\) is waiting for it (i receives a \textit{request} or a \textit{grant} message from \(j\)). In the same way, \(\text{delInArc}_i(j)\) models that \(i\) starts making what \(j\) awaited (i sends a \textit{grant} or a \textit{release} to \(j\)) while \(\text{delOutArc}_i(j)\) models that \(i\) stops waiting for \(j\) (i receives a \textit{grant} or a \textit{release} message). The \(\text{abort}_i\) action models that node \(i\) is aborted by the DDRA. Note that the abortion could remove several wait-for relations.

The execution of a \textit{discardMsg} action models the reception of obsolete messages (i.e. a grant message sent to an aborted process).

Our model may seem too abstract. However, [5] shows that it captures the dynamics of other less abstract models as the request-grant-release-withdraw model [9, 13].

3 Problem specification

The dynamics of the system could lead to a situation where a set of processes is indefinitely blocked waiting for resources held by other processes in the same set. That is,
the system can reach a state in which a set of nodes is dead-
locked.

In order to properly speak about the connection between
wait-for relations and deadlock, we will define the concepts
of path and cycle.

We define a path as a transitive wait-for relation between
two nodes.

**Definition 3.1 (Path).** We say that there is a path between
a node $n_i$ and a node $n_j$ at a state $s$ of the system, denoted
$n_i \xrightarrow{s} n_j$, if and only if $\exists n_1, \ldots, n_j \in N$ such that $n_i \rightarrow_{s} n_{i+1} \land n_i \in s.i_{n_{i+1}} \land \ldots \land n_j \in s.o_{n_{j-1}} \land n_{j-1} \in s.i_{n_j}$.

As a particular case, if $n_j \in s.o_{n_i} \land n_i \in s.i_{n_j}$, we
denote the path by $n_i \xrightarrow{s} n_j$.

On the path $n_i \xrightarrow{s} n_j$ we say that node $n_i$ is a predeces-
sor of node $n_j$ and that node $n_j$ is a successor of node $n_i$.

In a similar way, on the path $n_i \xrightarrow{s} n_j$ we say that node $n_i$ is an immediate predecessor of node $n_j$ and that node $n_j$ is an immediate successor of node $n_i$.

In the following we define a cycle as a particular kind of
path in which every node waits for itself transitively.

**Definition 3.2 (Cycle).** We say that there is a cycle at a state
$s$ of the system if and only if $\exists x \in N$ such that $x \xrightarrow{s}$ $x$.

We denote by cycles$(s)$ to the set of cycles in the system
at state $s$.

In the SR model, processes wait for at most one
resource, a deadlock is characterized by the existence of a
cycle in a graph representing the wait-for relations [8].

Given that the dynamics of the system could lead to a
deadlock and that a deadlock is a stable situation, it is obvi-
ous that an algorithm for deadlock resolution is necessary.

Such an algorithm must satisfy the following correctness
criteria:

(Liveness) If there is a cycle $C$ in the system then $\exists x \in$
nodes$(C)$ such that the abort$_x$ action is eventually exe-

cuted.

(Safety) The abort$_x$ action is enabled at a state $s$ of the
system if and only if node $x$ is deadlocked at state $s$, that is,
$x \xrightarrow{s} x$.

4 The distributed deadlock resolution algo-

rithm (DDRA)

In the following, we propose a low-communication-
cost distributed deadlock detection/resolution algorithm
that verifies the above criteria. First, in section 4.1, we
introduce the key idea of the DDRA. Later, in section 4.2, we
describe the algorithm through an execution example.

We use an Input/Output Automaton [10], denoted $S$, to
formally describe the proposed DDRA (see figures 1 and 2).

4.1 Key idea of the DDRA

The algorithm ensures that each node $x$ on a path receives
at most two messages. One of them informs about the
first predecessor of $x$ greater than $x$ (the closest prede-
cessor of $x$ greater than $x$) while the other informs about the
first successor of $x$ greater than $x$ (the closest successor of
$x$ greater than $x$).

In this way, only the second greatest node in a cycle can
receive two messages that inform about the same node (the
greatest node in the cycle). Therefore, by sending one of
those messages, this node can notify to the greatest node in
the cycle that it has to abort in order to resolve a deadlock.

Given that each node in a cycle receives at most two
messages with information about other nodes in the cycle
(one message in each direction), the algorithm has a mes-
joke complexity of $O(n)$.

4.2 Execution example

Next, we describe the proposed algorithm through an ex-
cution example. Consider the state of the system depicted
in the figure 3 where all the nodes are blocked.

**Initiation** When a node $x$ is blocked, it could be part
of a deadlock. In order to try to resolve the hypothetical
deadlock, the node initiates its computation (it executes the
initiate$_x$ action). Thereafter, we say that node $x$ is working
(status$_x$ = working).

A node $x$ initiates its computation by sending its current
instance $1$ in BM messages (backward messages) to all its
immediate predecessors that are less than $x$ and by sending
its current instance in an FM message (forward message)
to its immediate successor if this immediate successor is
less than $x$. The messages sent in the initiation process are
depicted in figure 3.a. Since node 1 is the smallest node,
it does not send any message; however, it is the destina-
tion of a BM message containing information about node 2
(its immediate successor) and on of an FM message contain-
ing information about node 5 (its immediate predecessor).
Note also that node 6, the greatest node in the cycle, sends
a BM message to node 2 and an FM message to node 4
with information about itself, but it is not the destination of
any message.

As can be observed, the initiation process satisfies the
main strategy of the algorithm: each node $x$ on a path only
receives information about the first node that is greater than
$x$ in each direction on the path.

---

1 An instance is a 2-tuple composed of the node identifier that initi-
ates the instance and the logical time at which the instance was created
(tblock).
DDRA useful definitions:
Instances \(\equiv\{(\text{initiator}, \text{tblock}) : \text{initiator} \in N, \text{tblock} \in N\}\)

Messages of the system model:
WAIT //The source node indicates that it has started waiting for the destination node
UNWAIT //The source node indicates that the wait-for relation with destination node has finished

Messages of the DDRA:
BM \(\equiv\{(fsg, dst, tblock)\} \)/(Backward Message
fsg \(\in\) Instances //instance of first successor greater (fsg) than destination node
dst, tblock \(\in\) Z //tblock of destination node
FM \(\equiv\{(fpg, dst, tblock)\} \)/(Forward Message
fpg \(\in\) Instance //instance of first predecessor greater (fpg) than destination node
dst, tblock \(\in\) Z //tblock of destination node

State (s):
A state \(s \in states(S)\), is defined by the following variables:
\(\forall i \in N : \text{in}_i \subseteq N\) // Set of nodes that are waiting for node \(i\)
\(\forall i \in N : \text{out}_i \subseteq N\) // Node to which node \(i\) is waiting
\(\forall i \in N : t\text{block}_i \in N\) //Logical time at which node \(i\) last blocked
\(\forall i \in N : fpg \subseteq\) Instances //Instances of first predecessors greater (fpg) than node \(i\)
\(\forall i \in N : fsg \subseteq\) Instances //Instances of first successors greater (fsg) than node \(i\)
\(\forall i \in N : status_i = \{\text{active, blocked, working, aborted}\}\) //The status of the node \(i\)
\(\forall i \in N : \text{frwrd}_i \subseteq\{(fsg, fpg, forw) : fsg, fpg \in \text{Instances}, forw \in \{\text{true, false}\}\}\) //Forwarded information
\(\forall i, j \in N, i \neq j : ch(i, j)\) is a FIFO queue // Communication channel between node \(i\) and \(j\)

Initial State \((s_0)\):
\(\forall i \in N : s_0, \text{ini}_i \leftarrow \emptyset\)
\(\forall i \in N : s_0, \text{out}_i \leftarrow \emptyset\)
\(\forall i \in N : s_0, \text{tblock}_i \leftarrow 0\)
\(\forall i \in N : s_0, fpg \leftarrow \emptyset\)
\(\forall i \in N : s_0, fsg \leftarrow \emptyset\)
\(\forall i \in N : s_0, \text{status}_i \leftarrow \text{active}\)
\(\forall i \in N : s_0, \text{frwrd}_i \leftarrow \emptyset\)
\(\forall i, j \in N, i \neq j : s_0, \text{ch}(i, j) \leftarrow \emptyset\)

Signature:
Input\((S)\) = \(\emptyset\)
Output\((S)\) = \{addOutArc\(_i\)(j), addInArc\(_i\)(j), delInArc\(_i\)(j), delOutArc\(_i\)(j), abort\_i : i, j \in N, i \neq j\}
Internal\((S)\) = \{\text{initiate}_i, rcvBM\_i, rcvFM\_i, frwrdMsg\_i, discardMsg\_i : i \in N\}

\(^{a}\)Note that, as we deal with the SR model, out\(_i\) is composed of at most by one element. We define it as a set to simplify the notation.

Figure 1. State, initial state \((s_0)\) and signature of the system \(S\).

Message forwarding A node \(k\) gathers information about greater nodes by the execution of the \(rcvBM\_k\) and the \(rcvFM\_k\) actions. An instance \((y, t_y)\) received in a BM message is interpreted by \(k\) as the existence of a path \(k \rightarrow y\). In a similar way, an instance \((x, t_x)\) received in an FM message is interpreted by \(k\) as the existence of a path \(x \rightarrow k\). Therefore, when node \(k\) has received a message in each direction (a BM message including an instance \((y, t_y)\) and an FM message including an instance \((x, t_x)\)), it knows that no node on the path \(x \rightarrow k \rightarrow y\) is greater than \(x\) and \(y\). Then, node \(k\) forwards the message about the greater node to the lesser node by the execution of the \(frwrdMsg\_k\) action. This operation mode ensures the desired behavior.

In figure 3.b, it can be observed that node 1, that has previously received a BM message with information about node 2 and an FM message with information about node 5, forwards the FM message with information about node 5 to node 2. In the same way, node 3 forwards the BM message because 5 is greater than 4.

In figure 3.c, once that node 2 has received the FM message with information about node 5 and the BM message with information about node 6, node 2 forwards to node 5 the FM message with information about node 6.

Detection and resolution Considering the message forwarding mechanism previously described, only the second greatest node in a cycle can receive the same instance (the instance of the greatest node in the cycle) in a BM and in an FM message (in both directions of the cycle). In this situ-
addOutArcs(j)
precondition:
status_i ≠ aborted ∧ out_i = ∅.
effects:
ch(i, j) ← ch(i, j) • WAIT;
out_i ← out_i ∪ j;
status_i ← blocked.

addInArcs(j)
precondition:
ch(j, i) = WAIT • γ.
effects:
ch(j, i) ← γ;
if (status_j = aborted)
ch(i, j) ← ch(i, j) • UNWAIT;
else
in_i ← in_i ∪ j;
if (status_j = working ∧ i > j)
ch(i, j) ← ch(i, j) • BM((i, tblock_i), -1).

delOutArcs(j)
precondition:
j ∈ in_i ∧ (out_i = ∅ ∨ ch(j, i) = UNWAIT • γ).
effects:
ch(j, i) ← γ;
out_i ← ∅;
status_i ← active;
tblock_i ← tblock_i + 1;
fsg_i ← ∅;
∀ fpg_j ∈ fpg_i : (fpg_i.initiator = j)
fpg_j ← fpg_j • fpg;
∀ (pre, suc, forw) ∈ frwrd_i : pre = fpg
frwrd_i ← frwrd_i - (pre, suc, forw).

delInArcs(j)
precondition:
j ∈ out_i ∧ ch(j, i) = UNWAIT • γ.
effects:
ch(j, i) ← γ;
out_i ← ∅;
status_i ← aborted;
tblock_i ← tblock_i + 1;
fsg_i ← ∅;
∀ fpg_j ∈ fpg_i : (fpg_i.initiator = j)
fpg_j ← fpg_j • fpg;
frwrd_i ← ∅.

initiate_i
precondition:
status_i = blocked.
effects:
status_i ← working;
∀ j ∈ out_i : i > j
ch(i, j) ← ch(i, j) • FM((i, tblock_i), -1);
∀ j ∈ in_i : i > j
ch(i, j) ← ch(i, j) • BM((i, tblock_i), -1).

recvBM_i
precondition:
ch(k, i) = BM(fsg_i, tblock_i) • γ
∧ ((k ∈ out_i ∧ fsg_i.initiator = k) ∨ tblock_i = tblock_k)
∧ fsg_i.initiator ≠ i.
effects:
ch(k, i) ← γ;
fsg_i ← fsg_i • fsg;
∀ fpg ∈ fpg_i
frwrd_i ← frwrd_i ∪ (fpg, fsg_i, false).

recvFM_i
precondition:
ch(k, i) = FM(fpg_i, tblock_i) • γ
∧ ((k ∈ in_i ∧ fpg_i.initiator = k) ∨ tblock_i = tblock_k).
effects:
ch(k, i) ← γ;
fpg_i ← fpg_i • fpg_i;
∀ fsg ∈ fsg_i
frwrd_i ← frwrd_i ∪ (fpg_i, fsg_i, false).

frwrdMsg_i
precondition:
∃ (fpg, fsg, forw) ∈ frwrd_i : forw = false.
effects:
frwrd_i ← frwrd_i - (fpg, fsg, forw);
∀ tblock_j ∈ tblock_i : ch(i, j) = ch(i, k) • BM(fpg_i, fsg_i, tblock_i);
∀ tblock_j ∈ tblock_i : ch(i, j) = ch(i, k) • FM(fpg_i, fsg_i, tblock_i).

abort_i
precondition:
ch(k, i) = BM(fsg_i, tblock_i) • γ
∧ tblock_i = tblock_k ∧ fsg_i = (i, tblock_i).
effects:
ch(k, i) ← γ;
∀ j ∈ in_i ∪ out_i
ch(i, j) ← ch(i, j) • UNWAIT;
in_i ← ∅;
out_i ← ∅;
status_i ← aborted;
tblock_i ← tblock_i + 1;
fsg_i ← ∅;
fpg_i ← ∅;
frwrd_i ← ∅.

discardMsg_i
precondition:
ch(k, i) = m • γ
∧ (no other action can consume the message).
effects:
ch(k, i) ← γ.

Figure 2. Actions of the system S.

As can be observed in figure 3. {a,b,c}, applying the described message forwarding strategy, the system reaches a state in which each node in the cycle except the greatest one (node 6) has received a message containing information about its first greater successor and predecessor in the cycle. Particularly, node 5, the second greatest node in the cycle, has received a BM message and an FM message containing information about node 6. Then, we are in the especial situation in which a node has received information about the same node in both directions. Therefore, node 5 forwards
Let a BM message to node 6 with information about node 6. When node 6 receives this BM message aborts and resolves the deadlock.

5 Correctness proof

The complete correctness proof of the algorithm is rather long. We only offer a sketch of how to prove both safety and liveness criteria.

5.1 Safety

In this section we prove that the algorithm satisfies the safety criterion. First, we define the concepts of instance, up-to date instance and relevant message which are used to understand the key lemma (lemma 5.4) to prove the safety theorem.

Definition 5.1 (instance). Let \( x \in \mathbb{N} \) and let \( t \in \mathbb{N} \). We say that \( (x, t) \) is an instance of node \( x \) and we denote it by \( I_x \).

In order to differentiate between previous instances of a node and the current instance of a node, we define the concept of up-to date instance.

Definition 5.2 (up-to date instance). Let \( (x, t) \) be an instance of the algorithm. We say that such an instance is up-to date at a state \( s \) of the system, denoted \( I_x(s) \), if and only if \( t = s.tblock_x \).

The algorithm discards messages related to outdated information. In order to properly speak about the messages that the algorithm consider as valid, we define the concept of relevant message.

Definition 5.3 (relevant message). Let \( m \) be a message of the algorithm. \( \forall i, j, k \in \mathbb{N}, \forall t \in \mathbb{Z} \), we say that \( m \) is a relevant message at a state \( s \) of the system if and only if anyone of the following situations holds:

i.1) \( m \in s.ch(j, i) \land m = BM(I_j, t) \land t = s.tblock_i \)

i.2) \( m \in s.ch(j, i) \land m = BM(I_j, t) \land t = s.tblock_{i} \)

ii.1) \( m \in s.ch(i, j) \land m = FM(I_i, t) \land i \in s.in_j \)

ii.2) \( m \in s.ch(k, j) \land m = FM(I_i, t) \land t = s.tblock_j \)

Given that not relevant messages are discarded when they arrive to their destination and that only messages containing up-to date instances promote to abortion, we concentrate our discussion on up-to date instances and relevant messages.

A blocked node \( j \) sends a BM message with its instance to each one of its predecessors \( k \) verifying \( k < j \). If this message continues to be relevant and the instance that it contains is up-to date, it means that \( j \) has not got unblocked, and therefore, \( k \) remains waiting for \( j \). Note that the reception of such a message includes the instance of \( j \) in the set \( fs_{gk} \). In a similar way, a blocked node \( i \) sends an FM message with its instance to its successor \( k \) if \( i > k \). If this message remains being relevant and the instance that it contains is up-to date, it means that \( i \) remains waiting for \( k \). Note also that the reception of such message includes the instance of \( i \) in the set \( fp_{kg} \). The following lemma generalizes these facts. It indicates that the existence of up-to date instances in sets \( fs_{gk} \) and \( fp_{kg} \), as well as the existence of relevant messages that contain up-to date instances traveling towards \( k \), ensures the existence of a path connecting \( k \) with the initiator of the instance and an order relation among the nodes of such a path.

Lemma 5.4. Let \( s \) be a reachable state of \( S \). \( \forall i, j, k \in \mathbb{N}, \forall t \in \mathbb{N} \), the following properties hold:

i.1) \( BM(I_j(s), t) \in s.ch(j, i) \land j \in s.out_i \Rightarrow \{ i \nrightarrow j \land j > i \} \)

i.2) \( I_j(s) \in s.fsg_i \Rightarrow \{ i \nrightarrow j \land j > i \land \max(nodes(i, j) - \{i, j\}) \} \)

i.3) \( BM(I_j(s), s.tblock_i) \in s.ch(k, i) \Rightarrow \{ i \nrightarrow j \land j > i \land \max(nodes(i, j) - \{i, j\}) \} \)
Lemma 5.6. Let \( \alpha = s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots \) be an execution of \( S \). The following properties hold:

i) \( \exists z_0 \text{ such that } \forall z \geq z_0 : \)
\[
\begin{align*}
&\{ i \not\in x \wedge i > j \} \\
&\{ j > \max(\text{nodes}(i - \frac{1}{n}, j) - \{i, j\}) \}
\end{align*}
\]
\( \Rightarrow \exists z_m \geq z_0 \text{ such that } \forall z_n \geq z_m : I_j(s_{z_n}) \in s_{z_n}.fsg_i \)

ii) \( \exists z_0 \text{ such that } \forall z \geq z_0 : \)
\[
\begin{align*}
&\{ i \not\in x \wedge i > j \} \\
&\{ j > \max(\text{nodes}(i - \frac{1}{n}, j) - \{i, j\}) \}
\end{align*}
\]
\( \Rightarrow \exists z_m \geq z_0 \text{ such that } \forall z_n \geq z_m : I_i(s_{z_n}) \in s_{z_n}.fpg_j \)

Proof. This lemma can be proved by induction over the length of a path. It is not difficult to show that it holds for a path of length 1. Then, when a node \( k \) knows the up-to date instance of its first predecessor greater than \( i \), \( j \), the execution of the action \( frwdMs_{sg_k} \) joins the knowledge about the path \( i \rightarrow k \) and the path \( k \rightarrow j \) leading to knowledge about the path \( i \rightarrow j \).

The liveness of the algorithm is concluded as a corollary of the previous lemma.

Theorem 5.7 (Liveness). \( S \) verifies the liveness criterion.

Proof. Let \( \alpha = s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots \) be an execution of \( S \). If \( C \in \text{cycles}(s_z) \), it is obvious that \( \exists j, k \in \text{nodes}(C) \) such that
\[
\begin{align*}
&s_z \rightarrow j \rightarrow k \rightarrow \max(\text{nodes}(\frac{1}{n}, j) - \{j, k\}) \\
&s_z \rightarrow k \rightarrow j \rightarrow \max(\text{nodes}(\frac{1}{n}, j) - \{j, k\})
\end{align*}
\]
and therefore,
\[
\begin{align*}
&j > k > \max(\text{nodes}(\frac{1}{n}, k) - \{j, k\}) \\
&j > k > \max(\text{nodes}(\frac{1}{n}, j) - \{j, k\})
\end{align*}
\]

So, by lemma 5.6, an up-to date instance of node \( j \) is eventually included in \( fpg_k \) and \( fsg_k \). That is, \( \exists z_1 \geq z \) such that
\[
\begin{align*}
&\forall z_2 \geq z_1 : \{ I_j(s_{z_2}) \in s_{z_2}.fpg_k \wedge \\
&I_j(s_{z_2}) \in s_{z_2}.fsg_k
\end{align*}
\]
Then, it can be shown that, by the execution of the action \( frwdMs_{sg_k} \), \( \exists z_3 \geq z_1 \) such that \( BM(I_j(s_{z_3}), s_{z_3}.tblock_j) \in s_{z_3}.ch(k, j) \). As the \( BM \) message is relevant and the instance it contains is up-to-date, the \( abort_j \) action will eventually be executed. That is, \( \exists z_n > z_3 \) such that \( \pi_{z_n} = abort_j \) and therefore, \( S \) verifies the liveness criterion.

6 Algorithm complexity

6.1 Communication cost

We claim that the proposed algorithm requires at most \( 2n - 1 \) messages in order to resolve a deadlock of \( n \) nodes. That is, it has a message complexity of \( O(n) \).

This low cost is achieved thanks to the main strategy that guides the algorithm.

Theorem 6.1 (Communication cost). When there is a deadlock composed of \( n \) nodes, the maximum number of messages needed to resolve it is \( 2n - 1 \).

Proof. Each node \( x \) in a cycle\(^2 \) can receive at most two relevant messages including an up-to-date instance of other nodes in the cycle: a \( BM \) message including the instance of the first successor of \( x \) greater or equal than \( x \) and an \( FM \) message including the instance of the first predecessor of \( x \) greater than \( x \). On the other hand, as there is not any predecessor of the greatest node greater than it is (among the nodes in the cycle), the greatest node will not receive any relevant \( FM \) message containing an up-to-date instance.

\(^2\)Note that a cycle is a particular kind of path.
Therefore, the maximum number of messages that the algorithm requires to resolve a deadlock of \( n \) nodes is \( 2n - 1 \) (\( nBM \) messages plus \( n - 1FM \) messages).

\[ \square \]

6.2 Latency

We claim that the proposed algorithm requires at most \( n \) messages sent in causal order to resolve a deadlock of \( n \) nodes. That is, it has a latency of \( O(n) \).

This low latency is achieved thanks to that neither the algorithm needs an extra round of messages to clean obsolete information as in [9] nor nodes need to restart their computation as in [4].

**Theorem 6.2** (Latency). When there is a deadlock composed of \( n \) nodes, the latency to resolve it is \( n \cdot T \), where \( T \) the inter-site communication delay.

**Proof.** Nodes collaborate in the graph exploration process. The path traversed by an instance is used (known and avoided) by a greater one. Therefore, the worst case latency to resolve a deadlock is related to a scenario in which the node that detects and resolves the deadlock does not get collaboration. This situation arises when the nodes are arranged in increasing or decreasing order. In this scenario, the message of the node that resolves the deadlock is handled by each node in the cycle, and therefore, it takes \( n \) steps.

\[ \square \]

7 Conclusions

In this paper, a new distributed algorithm for deadlock detection and resolution under the single-resource request model has been introduced.

The proposed algorithm has a message complexity \( O(n) \) that highly improves the communication cost of previous proposals, it keeps the lowest latency, \( O(n) \), and in addition, it is a safe-resolution algorithm (it only causes the strictly necessary abortions).

The algorithm achieves all these goals by using a new distributed mechanism that makes the nodes to collaborate in the graph exploration process.

In order to formally describe the algorithm and to provide a formal correctness proof, we have used the I/O Automata model as the basis for the formalism.

**Acknowledgments**

This work is supported by the Spanish Government under research grants numbers TIC2003-09420-C02-02 and TIS2004-02940.

**References**


