Guaranteed Deadlock Recovery: Deadlock Resolution with Rollback Propagation

Yi-Min Wang  Michael Merritt  Alexander B. Romanovsky

Abstract

Traditionally, deadlock resolution is performed by simply aborting any process or the lowest-priority process (called the victim) involved in a deadlock cycle. In message-passing applications where rollback propagation due to message dependencies is possible, the rollback of the victim may require other processes to roll back as well, and the restarted processes may get into the same deadlock again. We introduce the concept of guaranteed deadlock recovery which guarantees that a broken deadlock cycle will not be re-formed after the rollback, and show how to achieve this by carefully selecting the victim based on run-time dependency information. We also demonstrate a technique to incorporate a dynamic priority scheme into a distributed deadlock detection algorithm to perform guaranteed deadlock recovery.

1 Introduction

Checkpointing and rollback recovery is a technique that periodically saves the volatile state of a process onto stable storage so that the state can be restored when the process needs to roll back. Message logging is a technique that saves messages onto stable storage so that the messages can be replayed after a process rolls back to a checkpoint. Traditional checkpointing and message logging techniques [1-8] have been designed primarily for hardware failure recovery. A physical

*Yi-Min Wang and Michael Merritt are with AT&T Labs, Murray Hill, NJ 07974, USA. Alexander B. Romanovsky is with University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK.
failure uniquely determines the set of lost volatile process states and message logs. The nonvolatile checkpoints and message logs, the surviving volatile states and message logs, and the recorded message dependencies then uniquely determine the recovery line, indicating the set of processes that need to roll back and the states to which they should roll back to achieve consistent recovery with minimum rollback distance. In contrast, deadlock recovery does not involve any physical failure. When a deadlock is detected, a rollback is initiated by intentionally discarding some volatile states and message logs in order to allow the processes to take different execution paths to bypass the deadlock. Since rolling back any process (called the victim) involved in a deadlock cycle can break the cycle, we have the freedom to choose among multiple potential victims and hence multiple potential recovery lines. Our previous work on progressive retry [9, 10] applied the technique of checkpointing and message logging to recovering failed processes from software errors caused by unknown software bugs; message replaying and message reordering were employed as heuristics to bypass the software bugs. This paper shows that it is possible to guarantee error recovery for more specific types of errors such as deadlocks.

Much of the literature on deadlocks has focused on the detection problem for different resource request models [11–13]. After a deadlock is detected, resolution (or recovery) is usually performed by the detecting process simply aborting itself to release the resources it has held. A static priority can be assigned to each process so that the lowest-priority process involved in a deadlock cycle becomes the victim [14, 15]. In this paper, we consider deadlock resolution for message-passing applications in which processes communicate through interprocess messages as well as sharing resources through resource-related messages. We point out that the simple resolution method may not be sufficient for such systems because potential rollback propagation due to message dependencies may force other processes to roll back as well, and the temporarily broken deadlock cycle may reappear after the rollback. This motivates the concept of guaranteed deadlock recovery in which at least one process involved in a deadlock cycle can execute beyond the point of deadlock and so the same cycle will not reappear. For general nondeterministic executions, we identify a sufficient condition for guaranteed deadlock recovery in Section 3, that uses a directed rollback-dependency graph. Under the piecewise deterministic execution model [6, 16], we present in Section 4 a dynamic priority scheme based on transitive dependency tracking, and demonstrate a technique to incorporate it into a distributed deadlock detection algorithm so that the detecting
process has sufficient information to guarantee deadlock recovery. In the next section, we first
describe rollback-dependency and wait-for graphs.

2 Checkpointing and Deadlocks

2.1 Rollback-dependency graphs

In a message-passing application, the rollback of one process may require the rollback of other
processes in order to guarantee the consistency of system state. Specifically, if the sender of a
message $m$ “rolls back to a state before $m$ was sent” (i.e., unsend $m$), then the receiver of $m$ must
also “roll back to a state before $m$ was received” (i.e., unreceive $m$); otherwise, the states of the two
processes together would show that message $m$ has been received but not yet sent, which is clearly
inconsistent. Given a system with $N$ processes, a set of $N$ checkpoints, one from each process, is
called a global checkpoint. A consistent global checkpoint $T$ is a global checkpoint such that no
message is sent after a checkpoint of $T$ and received before another checkpoint of $T$ [17]. Figure 1(a)
gives an example checkpoint and communication pattern where each vertical solid bar represents a
nonvolatile checkpoint, each shaded triangle represents the current volatile process state (called
a volatile checkpoint), and each directed edge represents a message. Let $c_{i,x}$ ($0 \leq i \leq N - 1$ and
$x \geq 0$) denote the $x$th checkpoint of process $P_i$, where $i$ is the process id and $x$ is the checkpoint
index; let $I_{i,x+1}$ denote the interval between $c_{i,x}$ and $c_{i,x+1}$. Suppose the system in Figure 1(a)
decides to roll back to a consistent global checkpoint containing $c_{1,1}$ and $c_{2,2}$. Because message $g_{1,0}$
is unsent, process $P_0$ needs to roll back to $c_{0,1}$, which in turn unsend $m_{0,3}$ and forces $P_3$ to roll
back to $c_{3,1}$.

The above rollback propagation can be performed as a graph search on a rollback-dependency
graph [17] (or R-graph) as illustrated in Figure 1(b). Each node represents a checkpoint and an
edge is drawn from $c_{i,x}$ to $c_{j,y}$ if (1) $i = j$ and $y = x + 1$; or (2) a message is sent from $I_{i,x}$
and received in $I_{j,y}$. Such a dependency edge can be dynamically recorded at the receiver if the
sender piggybacks its process id and current checkpoint index on each outgoing message. When a
consistent global checkpoint needs to be computed, any process can collect the edge information
from all the other processes, construct the complete R-graph and perform rollback propagation by
doing a search on the graph. We have previously derived the following necessary and sufficient
condition for finding any consistent global checkpoint containing a target set of checkpoints [17]:

**Lemma 2.1** Given a target set of checkpoints $S$, a consistent global checkpoint containing $S$ exists if and only if, for every checkpoint $c_{i,x}$ of $S$ such that $c_{i,x+1}$ exists, $c_{i,x+1}$ does not reach any checkpoint of $S$ (denoted by $c_{i,x+1} \not\in S$) in the R-graph.

To minimize the rollback distance, the following algorithm can be used to compute the most recent consistent global checkpoint containing the target set [17]:

**Algorithm 1** Given a target set of checkpoints $S$, start an R-graph search from every $c_{i,x+1}$ such that $c_{i,x} \in S$, and mark every reachable node during the search. If any checkpoint of $S$ is marked, then it is not possible to find any consistent global checkpoint that contains $S$; otherwise, the last unmarked node of each process forms the most recent consistent global checkpoint containing $S$. 

---

Figure 1: (a) Example checkpoint and communication pattern; (b) rollback-dependency graph; (c) wait-for graph.
For example, Figure 1(b) shows the R-graph of Figure 1(a). To find the most recent consistent global checkpoint containing $c_{1,1}$ and $c_{2,2}$, the algorithm starts a search from $c_{1,2}$ and $c_{2,3}$. All the reachable nodes are marked to indicate that they must be rolled back, and the last unmarked node of each process ($\{c_{0,1}, c_{1,1}, c_{2,2}, c_{3,1}\}$) forms the desired global checkpoint.

2.2 Deadlocks and wait-for graphs

In this paper, we consider the one-resource model [11] in which each process can have at most one outstanding resource request at a time, and blocks its execution until the resource is granted. Figure 1(a) shows a resource access pattern in the form of message exchanges, that results in a deadlock. Notations are defined as follows: $\hat{P}_j$ stands for the resource manager for resource $R_j$; $r_{i,j}$ is a resource-request message sent from process $P_i$ to manager $\hat{P}_j$ to request exclusive access to $R_j$; $g_{j,i}$ is a resource-grant message sent from $\hat{P}_j$ to $P_i$; and $f_{i,j}$ is a resource-free message sent from $P_i$ to $\hat{P}_j$ to release the resource after its use. For simplicity, we assume that resources themselves do not have states, and each resource manager always has a checkpoint before every message-receiving event. This can be achieved by low-cost critical data checkpointing or by message logging under piecewise determinism [9].

For the purpose of presentation, we first assume that all resource-related messages are monitored by a central server. Distributed algorithms will be considered in a later section. The server maintains a wait-for graph (WFG) [11] as follows: a WFG-edge is drawn from $P_i$ to $\hat{P}_j$ if $P_i$ sends $r_{i,j}$ to $\hat{P}_j$ and resource $R_j$ is not available; an edge is drawn from $\hat{P}_j$ to $P_i$ if $\hat{P}_j$ sends $g_{j,i}$ to $P_i$ in which case $\hat{P}_j$ will be waiting for $P_i$ to release the resource; the edge is deleted from the WFG when $P_i$ sends $f_{i,j}$ to $\hat{P}_j$. The WFG-cycle in Figure 1(c) indicates the existence of a deadlock. Usually, a deadlock is resolved by aborting or rolling back one of the processes $P_k$ (called the victim) involved in the WFG-cycle so that the resources held by $P_k$ can be released and granted to other waiting processes.

3 Guaranteed Deadlock Recovery

In systems where processes sharing resources also communicate directly with each other via sending and receiving interprocess messages, deadlock recovery becomes more involved due to potential
rollback propagation. For example, in Figure 1, suppose the system decides to break the deadlock by rolling back \( \hat{P}_1 \) to \( c_{1,1} \) in order to reclaim the resource from \( P_0 \) and give it to \( P_3 \). Such an attempt may not be successful because rollback propagation along the interprocess message \( m_{0,3} \) forces \( P_3 \) to roll back to \( c_{3,1} \) and withdraw its resource request \( r_{3,1} \); as a result, the restarted \( \hat{P}_1 \) will still grant the resource to \( P_0 \) and the system is likely to get into the same deadlock situation again. In contrast, if the system instead chooses to roll back \( \hat{P}_2 \) to \( c_{2,1} \), the request \( r_{0,2} \) will remain valid when \( \hat{P}_2 \) is restarted. Process \( P_0 \) can then obtain the resource from \( \hat{P}_2 \) and proceed beyond the point of deadlock. (Even if there are other requests ahead of \( r_{0,2} \) in \( \hat{P}_2 \)'s request queue, \( \hat{P}_2 \) can properly reorder the messages to break the deadlock.) The example shows that deadlock recovery can be made more effective if rollback propagation due to message dependencies is taken into account in the victim selection process.

Motivated by the above example, we first introduce the concept of guaranteed deadlock recovery.

**Definition 1** A deadlock resolution algorithm is said to achieve guaranteed deadlock recovery if at least one of the processes involved in the WFG-cycle executes beyond the point of deadlock.

In general, after a process is rolled back, its reexecution may differ from the original execution in such a way that it becomes difficult to judge whether or not the execution has gone beyond the point of the original deadlock. Our approach to guaranteed deadlock recovery is to look for a resource that can be reclaimed and given to a non-rolled-back process. Specifically, among the deadlocked processes and resource managers, we attempt to roll back a resource manager to *unsend* its most recent resource-grant message without, at the same time, rolling back the process to which the reclaimed resource is supposed to be granted.

The following notation will be used throughout the paper. For each resource manager \( \hat{P}_j \) involved in a given WFG-cycle,

- \( g^j \) denotes the resource-grant message sent by \( \hat{P}_j \), corresponding to the outgoing WFG-edge of \( \hat{P}_j \);
- \( c^j \) denotes the checkpoint immediately before \( \hat{P}_j \) processed the resource-request message for which \( g^j \) was sent as a response;
- \( c^j_{\text{next}} \) denotes the checkpoint immediately after \( c^j \);
\( r^j \) denotes the resource-request message received by \( \hat{P}_j^i \), corresponding to the incoming WFG-edge of \( \hat{P}_j^i \) in the cycle;

- \( P^j \) denotes the sender of \( r^j \);

- \( v^j \) denotes the volatile checkpoint of \( P^j \).

For example, if we consider \( \hat{P}_j^2 = P_2 \) in Figure 1(a), then \( g^j = g_{2,3}, c^j = c_{2,1}, c_{\text{next}}^j = c_{2,2}, r^j = r_{0,2}, P^j = P_0 \) and \( v^j = c_{0,2} \). The problem of achieving guaranteed deadlock recovery can then be formulated as: given a WFG-cycle, we attempt to choose the victim to be a resource manager \( \hat{P}_j^i \) in the cycle such that a consistent global checkpoint \( T \) can be found to contain both \( c^j \) and \( v^j \).

We then have the following sufficient condition for guaranteed deadlock recovery.

**Theorem 1** Given a WFG-cycle, guaranteed deadlock recovery can be achieved if there exists a resource manager \( \hat{P}_j^i \) in the cycle such that \( c_{\text{next}}^j \not\to \{c^j, v^j\} \) in the R-graph.

**Proof:** Since \( v^j \) is a volatile checkpoint, the “next checkpoint” of \( v^j \) does not exist. If there exists \( \hat{P}_j^i \) such that \( c_{\text{next}}^j \not\to \{c^j, v^j\} \), then there exists a consistent global checkpoint \( T \) containing \( \{c^j, v^j\} \) by Lemma 2.1. Since \( c^j \) is a state in which \( \hat{P}_j^i \) has not granted its resource \( R_j \) to any process, and \( v^j \) is a state in which \( P^j \) is still waiting for \( R_j \), guaranteed deadlock recovery can be achieved by rolling back the system to \( T \) and forcing \( \hat{P}_j^i \) to grant \( R_j \) to \( P^j \).

For each checkpoint pair \( \{c^j, v^j\} \) corresponding to a \( \hat{P}_j^i \) in a given WFG-cycle, Algorithm 1 can be used to test the condition in Theorem 1 as well as computing the most recent consistent global checkpoint containing \( \{c^j, v^j\} \) when the condition is true. By starting a search from \( c_{\text{next}}^j \), if either \( c^j \) or \( v^j \) is marked, then the algorithm is aborted and the next pair is tested. If the search finishes for a given checkpoint pair, then the last unmarked checkpoints of the processes form the recovery line to guarantee deadlock recovery. If the search is aborted for every such checkpoint pair \( \{c^j, v^j\} \), then guaranteed deadlock recovery cannot be achieved. For example, Figure 2 shows the results of applying Algorithm 1 to the two checkpoint pairs in Figure 1 for deadlock recovery. In Figure 2(a), the search for the checkpoint pair \( \{c_{1,1}, c_{3,2}\} \) is aborted because \( c_{1,2} \) can reach \( c_{3,2} \) (through \( c_{0,2} \)). In contrast, Figure 2(b) shows that \( c_{2,2} \) can reach neither \( c_{2,1} \) nor \( c_{0,2} \), and so \( \{c_{0,2}, c_{1,2}, c_{2,1}, c_{3,0}\} \), identified after the search, can be used as the recovery line to guarantee deadlock recovery. Suppose
an edge from $c_{3,1}$ to $c_{0,1}$ is added because of an additional interprocess message. Then $c_{0,2}$ becomes reachable from $c_{2,2}$, and guaranteed deadlock recovery can no longer be achieved.

Figure 2: Algorithm 1 execution for checkpoint pairs (a) $\{c_{1,1}, c_{3,2}\}$ and (b) $\{c_{2,1}, c_{0,2}\}$.

4 Distributed Deadlock Recovery

4.1 Piecewise determinism

Much of the literature on rollback recovery is based on a model of piecewise determinism (PWD) [6,16]. Under the PWD assumption, each process execution is modeled as consisting of a number of state intervals bounded by message-receiving events. Execution within each state interval is completely deterministic and replayable. This allows the use of message logging as a form of checkpointing. Specifically, logging all the messages that have been processed since the most recent checkpoint equivalently places a logical checkpoint [9] at the end of current state interval (or, equivalently, just before the next message-receiving event) because of the capability of deterministic replay up to that point. Therefore, the piecewise deterministic model can be viewed as having a logical checkpoint before every message-receiving event. It has been shown that [17], under the PWD assumption, R-graph reachability can be tested in a distributed fashion if the following transitive dependency tracking is employed: each process $P_i$ maintains a size-$N$ transitive dependency vector $D_i$. The entry $D_i[k]$ is initialized to 1, incremented every time a new state interval starts, and so always represents the current state interval index. Every other entry $D_i[j]$, $j \neq i$, is initialized to 0 and records the highest index of any state intervals of $P_j$ on which $P_i$'s current
state transitively depends. When $P_i$ sends a message $m$, its current $D_i$ vector is piggybacked on $m$. When $P_j$ receives $m$, its $D_j$ vector is updated to be the coordinate-wise maximum of its current $D_j$ and the piggybacked $D_i$.

We next show in Theorem 2 that, under the PWD assumption, any resource manager can locally determine whether or not it is a good candidate for victim selection to guarantee deadlock recovery. The proof is based on the following lemma [17] which translates R-graph reachability into dependency vector comparison.

**Lemma 4.1** Under the PWD assumption, for any two logical checkpoints $c_{i,x}$ ($x \neq 0$) and $c_{j,y}$, $c_{i,x} \rightarrow c_{j,y}$ if and only if $D_{j,y}[i] \geq x$ where $D_{j,y}$ is the $D_j$ vector at state interval $I_{j,y}$ (which ends with the logical checkpoint $c_{j,y}$).

**Theorem 2** For a resource manager $P_j^*$ in a given WFG-cycle, let $Z$ denote the index of the state interval from which $g^j$ was sent, and let $D^j$ denote the transitive dependency vector piggybacked on $r^j$. Guaranteed deadlock recovery can be achieved if there exists a resource manager $P_j^*$ in the cycle such that

$$D^j[j] < Z. \tag{1}$$

**Proof:** We only need to prove that the condition $c^j_{next} \not\rightarrow \{c^j, v^j\}$ in Theorem 1 reduces to $D^j[j] < Z$ under the PWD assumption. From the definition of $Z$, $c^j = c_{j,Z-1}$ and $c^j_{next} = c_{j,Z}$. For ease of presentation, let $v^j = c_{k,v}$. Since $D_{j,Z-1}[j] = Z - 1 < Z$, $c^j_{next} \not\rightarrow c^j$ must always be true by Lemma 4.1. The condition $c^j_{next} \not\rightarrow v^j$ becomes $D_{k,v}[j] < Z$ by Lemma 4.1. Since process $P^j$ (the sender of $r^j$) must have sent $r^j$ from state interval $I_{k,v}$ and then blocked at volatile checkpoint $c_{k,v}$, we have $D^j = D_{k,v}$. Therefore, $D^j[j] = D_{k,v}[j] < Z$. \hfill \blacksquare

Each resource manager $P_j$ can test the condition in Eq. (1) by recording the index $Z$ of the state interval from which the most recent grant message was sent, and comparing that index against the $j^{th}$ entry of the dependency vector piggybacked on each request message to be queued. Also, when a resource is released and granted to another process, the resource manager will update $Z$, and so the test result corresponding to each of the remaining incoming WFG-edges may need to be dynamically updated.
4.2 Dynamic priorities

Most existing priority-based distributed deadlock detection algorithms allow only a predetermined static priority for each process, and the process with the lowest static priority in a cycle is aborted to resolve the deadlock [11, 12, 14]. The concept of guaranteed deadlock recovery suggests that a dynamic priority scheme which incorporates run-time information can help select a better victim to achieve more effective deadlock recovery. In this section, we present a dynamic priority-based algorithm which permits spontaneous priority changes at any time while preserving essential invariants for deadlock detection. This algorithm is a simple extension of a known static priority-based deadlock detection algorithm [14]. Such an extension provides a general dynamic priority scheme to take into account any recovery-related information such as degree of nondeterminism, estimated rollback cost, state dependencies, etc. We then describe how to manage the dynamic priority scheme based on Eq. (1) as a special case. For simplicity in the presentation and proof, the algorithms and proofs in this paper will detect the highest priority in a cycle, rather than the lowest. Modifications to detect the lowest priority process are straightforward.

Figure 3 presents an abstraction of the deadlock detection algorithm as a state-transition system. The algorithm sends probe messages in the opposite direction of WFG-edges and detects a cycle when a probe comes back to its initiator. Each probe message also collects the maximum priority of any node that it has visited so that the maximum-priority process of a WFG-cycle can abort itself to resolve the deadlock when it sees its own priority return. Each process has a public label, private label, public priority and private priority. Call the public label and priority the process’s public pair, and the private label and priority its private pair. Since the sets of labels and priorities are each totally ordered, we can use the natural lexicographic ordering on pairs (public label, public priority). The wait-for information is maintained and propagated by the five state transitions shown in Figure 3 and described below.

Block occurs when a resource-request or a resource-grant message creates a WFG-edge. Both the private and public labels of the waiting process are changed to a value greater than its old public label and greater than the public label of the process on which it is waiting, transmitted to the waiting process via a probe message. (The function inc(u, v) nondeterministically chooses a value greater than both u and v and unique to the waiting process.) The public priority of the waiting process is set to its own private priority. Transmit occurs when process $P$ is waiting for
process $Q$, and $P$ is informed (via a probe message) that the public label $v$ of $Q$ is greater than its own public label $u$, or $u = v$ and the public priority $q$ of $Q$ is greater than its own public priority, $r$. Process $P$ then changes its public label to $v$ and sets its public priority to the maximum of $q$ and its own private priority, $p$. Detect occurs when process $P$ is waiting for process $Q$, and process $P$ detects that the public label of $Q$ is the same as its own and the public priority of $Q$ is the same as its private priority. The probe message must have propagated the highest public label and picked up the maximum private priority in its first trip around the entire cycle, and then propagated the maximum private priority to its owner (i.e., the detecting process) in the second trip. The algorithm guarantees that the process with the maximum priority among all processes in the cycle will execute Detect. Activate occurs when one process stops waiting for another: a resource may be released, the waiting process may time-out or be aborted by deadlock resolution. The detection algorithm places no precondition on this transition (other than the existence of the WFG edge). This allows maximum nondeterminism in allowing for spontaneous aborts (due, for example, to timeouts), and in allowing for arbitrary deadlock resolution strategies. (For example, in the resolution scheme we discuss below, the detecting process is not the process that actually
aborts.) Change\_P can be executed by any process at any time to change its priority. The function inc\((u)\) nondeterministically chooses a value greater than \(u\) and unique to the process.

This algorithm extends an algorithm due to Mitchell and Merritt, [14], by adding the Change\_P transition to allow priorities to change dynamically.

### 4.2.1 Correctness

In this section, we summarize the important properties of the above algorithm in two theorems and prove their correctness. To make the presentation clear, the proofs of the lemmas are given in the Appendix.

**Theorem 3** If a cycle of \(n\) processes forms and persists long enough, then a process in the cycle will execute the Detect step within \(2n - 2\) Transmits of the last Change\_P in the cycle.

**Proof:** The proof is essentially identical to the corresponding proof of the original algorithm [14].

The second correctness property indicates that processes do not execute the Detect step without reasonable justification. (Aborting every waiting process avoids deadlock, but also avoids progress!) We are able to show that any process that executes the Detect step is the maximum-priority process of a cycle that could have happened, in the sense that the cycle exists in a global state of an execution of the algorithm that is indistinguishable to the processes from the original execution. As with the original deadlock algorithm, this one can be modified to Detect only cycles that exist in a global state of the WFG, by sending another message around the cycle after the Detect step. However, this produces a delay of \(n\) steps to detect the deadlock, to little real advantage [14]. Instead, the algorithm only detects cycles that existed in an equivalent virtual run of the system, in a sense we make precise below.

We consider an occurrence of a state transition of the algorithm to be an *event*, and model an execution of the algorithm as a (finite or infinite) sequence of events, which we will call a *run*. The events record enough information about the execution to unambiguously reconstruct the state of the processes and the wait-for graph at each point. Hence, let \(WFG(\alpha)\) denote the wait-for graph on processes after run \(\alpha\). Specifically, looking at Figure 3, an occurrence of the Block transition would be denoted \(Block(P,Q,u',p)\), where \(u'\) is the result returned by \(inc(u,v)\). Occurrences of the other
transitions would be denoted \(\text{Transmit}(P,v,p')\), where \(p' = \max(p,q)\), \(\text{Detect}(P)\), \(\text{Activate}(P)\), and \(\text{Change}_P(u',q)\), where \(u'\) is the value returned by \(\text{inc}(u)\). Each of these events is said to occur at \(P\).

If \(\pi\) is an event of process \(P\) in run \(\alpha\), let \(\text{PubL}(\pi,\alpha)\) be the public label of \(P\) immediately after \(\pi\) in \(\alpha\), let \(\text{PubP}(\pi,\alpha)\) be the public label of \(P\) immediately after \(\pi\) in \(\alpha\), and let \(\text{Pub}(\pi,\alpha)\) be \((\text{PubL}(\pi,\alpha),\text{PubP}(\pi,\alpha))\). Define \(\text{PrivL}(\pi,\alpha)\), \(\text{PrivP}(\pi,\alpha)\), and \(\text{Priv}(\pi,\alpha)\) similarly to be the private label, private priority, and private pair of \(P\) immediately after \(\pi\) in \(\alpha\).

Similarly, for process \(P\) and run \(\alpha\), let \(\text{PubL}(P,\alpha)\), \(\text{PrivL}(P,\alpha)\), \(\text{PubP}(P,\alpha)\), \(\text{PrivP}(P,\alpha)\), \(\text{Pub}(P,\alpha)\), and \(\text{Priv}(P,\alpha)\) denote the public and private labels, public and private priorities and public and private pairs, respectively, at \(P\) after \(\alpha\).

Call instances of the \(\text{Block}\), \(\text{Change}_P\), and \(\text{Transmit}\) transitions the \textit{mutating} events in a run. Note that these are the only events that change a process's public label or priority, and that they always increase the public pair value.

**Lemma 4.2**

1. In any run, no process's public label is ever smaller than its private label, and its public priority is never smaller than its private priority.

2. In any run, the private pairs at any process form a non-decreasing sequence.

3. In any run, the public pairs at any process form a non-decreasing sequence.

4. Let \(\alpha\) be a run and \(u\) be a public label of some process in \(\alpha\).

   (a) If \(u\) is the initial public label of some process \(P\) in \(\alpha\), then \(p_u = \min\{p|(u,p)\}\) is a public pair in \(\alpha\}, where \(p_u\) is the initial public priority of \(P\).

   (b) If \(u\) is not the initial public label of any process in \(\alpha\), then there is a unique process, \(P\), and Block or \(\text{Change}_P\) event \(\pi\) at \(P\) in \(\alpha\), such that \(\text{PubL}(\pi,\alpha) = u\). Moreover, \(\text{PubP}(\pi,\alpha) = \min\{p|(u,p)\}\) is a public pair in \(\alpha\}\).

If \(\pi\) is an event of process \(P\) in run \(\alpha\), let \(\text{Pub}(\pi,\alpha)\) be the ordered pair consisting of the public label and the public priority of \(P\) immediately after \(\pi\) in \(\alpha\). Call instances of the \(\text{Block}\), \(\text{Change}_P\),
and Transmit transitions the *mutating* events in a run. Note that these are the only events that change a process's public label or priority, and that they always increase the public pair value.

For any run $\alpha$, define the *direct-cause relation*, $DC(\alpha)$, on the events in $\alpha$ as follows. If $\pi$ is an event of process $P$ in $\alpha$, then:

1. $(\phi, \pi) \in DC(\alpha)$ if $\phi$ is an event of $P$ that occurs earlier than $\pi$ in $\alpha$.
2. $(\phi, \pi) \in DC(\alpha)$ if $\pi$ is a Transmit or Detect event where $P$ is blocked on process $Q$, and $\phi$ is the last mutating event of $Q$ that precedes $\pi$ in $\alpha$, if such an event exists.
3. $(\pi, \phi) \in DC(\alpha)$ if $\pi$ is a Block, Transmit, or Detect event where $P$ blocks on process $Q$, and $\phi$ is the first mutating event of $Q$ following $\pi$ in $\alpha$, such that $Pub(\pi, \alpha) \leq Pub(\phi, \alpha)$, if such an event exists.

**Lemma 4.3** If $(\pi, \phi) \in DC(\alpha)$, then $Pub(\pi, \alpha) \leq Pub(\phi, \alpha)$.

Given a sequence of events $\alpha$ and process $P$, let $\alpha|P$ denote the subsequence of $\alpha$ consisting of events which occur at $P$.

**Lemma 4.4** Let $\alpha$ be a run and let $\gamma$ be any sequence that is a reordering of $\alpha$ consistent with $DC(\alpha)$. (That is, $\pi$ precedes $\phi$ in $\gamma$ if $(\pi, \phi) \in DC(\alpha)$.) Then $\gamma$ is a run, and for every process $Q$, $\alpha|Q = \gamma|Q$.

Now for any run $\alpha$, define the *causal relation*, $C(\alpha)$, on the events in $\alpha$ to be the transitive, reflexive closure of $DC(\alpha)$. Because $DC(\alpha)$ is a subset of the order of events in $\alpha$, $C(\alpha)$ is a partial order. Let $\alpha$ be a run and $\pi$ be an event in $\alpha$. Define the *causal closure* of $\pi$ in $\alpha$, $CC(\pi, \alpha)$, to be the subsequence of events in $\alpha$ that are ordered before $\pi$ by $C(\alpha)$.

**Lemma 4.5** Let $\alpha$ be a run and $\pi$ be an event in $\alpha$. Then $CC(\pi, \alpha)(\alpha - CC(\pi, \alpha))$ is a run.

**Theorem 4** If $\alpha\pi$ is a run ending in a Detect event $\pi$ at process $P$, then $CC(\pi, \alpha\pi)$ is a run, and there is a cycle of blocked processes in $WFG(CC(\pi, \alpha\pi))$ in which $P$ has the maximum priority.

**Proof:** By Lemma 4.5, $CC(\pi, \alpha\pi)$ is a run. Let $Pub(P, CC(\pi, \alpha\pi)) = (u, p)$. We will prove the following claim by induction on the length of the path from $P$ in $WFG(CC(\pi, \alpha\pi))$: for each node $Q$ reachable from $P$ in $WFG(CC(\pi, \alpha\pi))$: 14
- \( Pub(Q, CC(\pi, \alpha \pi)) = (u, p) \)

- if \( Q \neq P \), the last event of \( Q \) in \( \alpha \) is a Transmit and \( p \) is greater than the private priority of \( Q \).

- The farther \( Q \) is from \( P \) in \( WFG(CC(\pi, \alpha \pi)) \), \( Q \neq P \), the earlier it’s final Transmit step is in \( \alpha \pi \).

Since \( \alpha \pi \) is finite, by the claim, the path from \( P \) must eventually reach a process which reaches \( P \). Note also that if \( Q \) is reachable from \( P \) in \( WFG(CC(\pi, \alpha \pi)) \), \( Q \neq P \), then Lemma 4.2.1 implies that \( p \) is greater than the private priority of \( Q \), proving the theorem.

The induction basis is trivial, so let \( Q \) be a node reachable from \( P \) in \( WFG(CC(\pi, \alpha \pi)) \) by a path of at least one edge, and let \( P' \) be the predecessor of \( Q \) on the path from \( P \). If \( Q = P \), the claim and hence the theorem follow.

Assume \( Q \neq P \). If \( P' = P \), since \( \pi \) is the last event in \( CC(\pi, \alpha \pi) \), \( Pub(Q, CC(\pi, \alpha \pi)) = (u, p) \). If \( P' \neq P \), by induction the last event, \( \phi \), of \( P' \) in \( CC(\pi, \alpha \pi) \) is a Transmit event and \( Pub(P', CC(\pi, \alpha \pi)) = (u, p) \). By the definition of the Transmit transition and Lemma 4.3, \( Pub(Q, CC(\pi, \alpha \pi)) = (u, p) \).

Since \( p \) is the private priority of process \( P \), and \( P \neq Q \), \( p \) cannot be the private priority of process \( Q \). It follows that the last mutating event of process \( Q \) in \( CC(\pi, \alpha \pi) \) must be a Transmit event, \( \psi \). Moreover, since \( \alpha \pi \) is a run, if \( P' \neq P \), Lemma 4.2.3 implies that \( \psi \) must precede the final Transmit of \( P' \), \( \phi \), in \( \alpha \pi \).

Suppose \( \psi \) is not the last event of process \( Q \) in \( CC(\pi, \alpha \pi) \). Let \( \psi' \) be that last event of \( Q \). We know that \( \psi' \) is not a mutate event, and since \( p \) is not the private priority of process \( Q \), \( \psi' \) is also not a Detect event. Hence, \( \psi' \) must be an Activate event. By the definition of \( CC(\pi, \alpha \pi) \), \( (\psi', \pi) \in C(\alpha \pi) \). But from the definition of \( C(\alpha \pi) \), there exists an event \( \psi'' \) of \( Q \) such that \( (\psi', \psi'') \in DC(\alpha \pi) \) and \( (\psi'', \pi) \in C(\alpha \pi) \). But then \( \psi' \) is not the last event of \( Q \) in \( CC(\alpha \pi) \), a contradiction.

4.2.2 Dynamic priorities for guaranteed deadlock recovery

We now describe a dynamic priority scheme based on Eq. (1). The unique private priority of each process is considered to have the form (dynamic priority, static priority, process id), and the
lexicographic ordering is used for comparison. Given a pair \((P_k, P_j)\) where process \(P_k\) is waiting for \(P_j\) to grant its resource, we call the pair a \(G\)-pair (\(G\) stands for “Guarantee”) if it satisfies Eq. (1); otherwise, the pair is called an \(NG\)-pair. Our approach is to assign a greater priority to \(G\)-pairs so that a process belonging to a \(G\)-pair will preferentially detect the deadlock and perform guaranteed deadlock recovery. Since a resource manager may have multiple waiting processes and hence may belong to several \(G\)-pairs and \(NG\)-pairs, our scheme associates the priority of a pair with the process, instead of the manager. The dynamic priority has one of three values: low, medium, and high. The dynamic priority of every resource manager is always assigned low. A non-manager process has a medium priority if it belongs to an \(NG\)-pair, and a high priority if it belongs to a \(G\)-pair. Since resource managers never wait directly for each other, all cycles contain non-manager processes and the detecting process can never be a manager.

When process \(P_k\) executes the Detect step, it notifies the manager \(P_j\) that it is waiting for and selects \(P_j\) to be the victim. Manager \(P_j\) is required to initiate a rollback to reclaim its resource and give that resource to \(P_k\). If there is at least one \(G\)-pair in the WFG-cycle, then the detecting process \(P_k\) must have a high priority and guaranteed deadlock recovery can be achieved; otherwise, \(P_k\) has a medium priority and guaranteed deadlock recovery cannot be achieved.

Processes change their priorities in two circumstances. First, when a resource manager \(P_j\) receives a request message and the resource is not available, the requesting process updates the process priority on-line by executing a \texttt{Change\_P} immediately following a \texttt{Block}. (The probe message sent by \(P_j\) needs to contain the test result of Eq. (1) as well, so that the requesting process can set its dynamic priority accordingly.)

The second circumstance is used to update the priorities of the remaining waiting processes on-line when a resource is released to \(P_j\) and then granted to another process. This requires a \texttt{Change\_P} at some of the remaining waiting processes immediately after \(P_j\) executes an \texttt{Activate} followed by a \texttt{Block}. We note that the \texttt{Activate} step of \(P_j\) is triggered by receiving a resource-free message which starts a new state interval. Therefore, the \texttt{Block} step after the \texttt{Activate} must correspond to a grant message sent from a new state interval with an index higher than that of any existing interval. In other words, the new value of \(Z\) must be greater than the \(D_i[j]\) entry of every existing request message. Hence, every existing \(NG\)-pair involving \(P_j\) becomes a \(G\)-pair and the waiting process in the pair needs to execute a \texttt{Change\_P}, while every existing \(G\)-pair remains unaffected.
5 Summary

This paper has introduced new research dimensions for both rollback recovery and deadlock resolution. From rollback recovery point of view, the recovery line is no longer uniquely defined and the freedom of choosing among multiple potential recovery lines allows application-specific information to be exploited for more effective recovery. From deadlock resolution point of view, the additional consideration of rollback propagation has motivated the concept of guaranteed deadlock recovery, and victim selection based on dependency information requires distributed deadlock detection algorithms to accommodate a dynamic priority scheme. For general nondeterministic executions, we have derived a sufficient condition for guaranteed deadlock recovery in terms of the reachability on a rollback-dependency graph. We have also shown that, under piecewise deterministic assumption, the condition can be locally tested by each resource manager based on transitive dependency information, and is therefore well-suited for distributed deadlock detection algorithms.

Appendix

In this section, we restate Lemmas 4.2–4.5 and present the proofs.

Lemma 4.2

1. In any run, no process’s public label is ever smaller than its private label, and its public priority is never smaller than its private priority.

2. In any run, the private pairs at any process form a non-decreasing sequence.

3. In any run, the public pairs at any process form a non-decreasing sequence.

4. Let α be a run and u be a public label of some process in α.

   (a) If u is the initial public label of some process P in α, then pu = min{p|(u,p) is a public pair in α}, where pu is the initial public priority of P.

   (b) If u is not the initial public label of any process in α, then there is a unique process, P, and Block or Change_p event π at P in α, such that PubL(π,α) = u. Moreover, PubP(π,α) = min{p|(u,p) is a public pair in α},
Proof:

1. A simple induction on the sequence of events.

2. A simple induction, observing that no action decreases the private pair value.

3. A simple induction, observing that no action decreases the public pair value.

4. Another induction, using the fact that private labels are unique to each node, and that no action decreases the public priority associated with any public label.

Lemma 4.3 If \((\pi, \phi) \in DC(\alpha)\), then \(Pub(\pi, \alpha) \leq Pub(\phi, \alpha)\).

Proof: Let \((\pi, \phi) \in DC(\alpha)\). The proof is a case analysis based on the definition of DC(\alpha):

1. \(\pi\) is an event of \(P\) that occurs earlier than \(\phi\) in \(\alpha\). By Lemma 4.2.3, \(Pub(\pi, \alpha) \leq Pub(\phi, \alpha)\).

2. \(\phi\) is a Transmit or Detect event where \(P\) is blocked on process \(Q\), and \(\pi\) is the last mutating event of \(Q\) that precedes \(\phi\) in \(\alpha\). Then \(Pub(\pi, \alpha)\) is the public pair at \(Q\) when \(\phi\) occurs. By the definition of the Transmit and Detect actions, \(Pub(\pi, \alpha) \leq Pub(\phi, \alpha)\).

3. \(\pi\) is a Block, Transmit, or Detect event where \(P\) blocks on process \(Q\), and \(\phi\) is the first mutating event of \(Q\) following \(\pi\) in \(\alpha\), such that \(Pub(\pi, \alpha) \leq Pub(\phi, \alpha)\).

Lemma 4.4 Let \(\alpha\) be a run and let \(\gamma\) be any sequence that is a reordering of \(\alpha\) consistent with \(DC(\alpha)\). (That is, \(\pi\) precedes \(\phi\) in \(\gamma\) if \((\pi, \phi) \in DC(\alpha)\).) Then \(\gamma\) is a run, and for every process \(Q\), \(\alpha|Q = \gamma|Q\).

Proof: Note that by the first case in the definition of \(DC(\alpha)\), \(\alpha|Q = \gamma|Q\).

The proof that \(\gamma\) is a run is an induction on the length of \(\gamma\), with a trivial basis.

For the induction step, let \(\beta\phi\) be a prefix of \(\gamma\) ending in a single event \(\phi\) of process \(P'\), and assume \(\beta\) is a run.

To show that \(\beta\phi\) is a run, we must show that \(\phi\) is enabled after \(\beta\). Let \(\alpha'\phi\) be the prefix of \(\alpha\) ending in the event \(\phi\). Since \(\gamma|P = \alpha|P\), \(\beta|P' = \alpha'|P'\). Hence, \(Pub(P', \beta) = Pub(P', \alpha')\), and there is an edge from \(P'\) to some process \(Q'\) in the \(WFG(\beta)\) if and only if such an edge exists in \(WFG(\alpha')\).

The remainder of the proof is a case analysis.
1. $\phi$ is a Change$_P$ event:
   That $\phi$ is enabled after $\beta$ follows because $\beta|P' = \alpha'|P'$.

2. $\phi$ is an Activate event:
   Let $Q'$ be the target of the edge from $P'$ in $WFG(\alpha')$. By the argument above, there is an edge from $P'$ to $Q'$ in $WFG(\beta)$, and hence $\phi$ is enabled in $\beta$.

3. $\phi$ is a Detect event over an edge to $Q'$:
   By the definition of this transition, $Pub(P', \alpha'\phi) = Pub(Q', \alpha'\phi)$. By the definition of the $DC$ relation, $\phi$ is preceded and followed by the same mutating events of $Q'$ in $\beta$ as in $\alpha$, and hence $Pub(Q', \alpha') = Pub(Q', \beta)$. It follows that $\phi$ is enabled after $\beta$.

4. $\phi$ is a Transmit event over an edge to $Q'$:
   Suppose first that $P'$ transmit's the priority from $Q$ in $\alpha'\phi$: that $PrivP(P', \alpha') \leq PubP(Q', \alpha')$.
   Then $Pub(P', \alpha'\phi) = Pub(Q', \alpha')$. As in the case above, by the definition of the $DC$ relation, $\phi$ is preceded and followed by the same mutating events of $Q'$ in $\beta$ as in $\alpha$, and hence $Pub(Q', \alpha') = Pub(Q', \beta)$. It follows that $\phi$ is enabled after $\beta$.

Now assume that in $\alpha'\phi$, $P'$ promotes its own private priority: that $PubP(Q', \alpha') < PrivP(P', \alpha')$.
Since $\phi$ is enabled after $\alpha'$, $Pub(P', \alpha') < Pub(Q', \alpha')$, and so $PubL(P', \alpha') < PubL(Q', \alpha')$.
To show that $\phi$ is enabled after $\beta$, it suffices to show that $Pub(P', \beta) < Pub(Q', \beta)$, that $PubL(P', \beta\phi) = PubL(Q', \beta)$, and $PubP(Q', \beta) < PubP(P', \beta\phi)$.

By the definition of the $DC$ relation, $\phi$ is preceded in $\beta$ by the mutating events of $Q'$ in $\alpha'$ (and possibly by some mutating events of $Q'$ that follow $\phi$ in $\alpha$).

By the induction hypothesis and Lemma 4.2.3, $Pub(P', \beta) = Pub(P', \alpha')$, and $Pub(Q', \alpha') \leq Pub(Q', \beta)$. Above we noted that $Pub(P', \alpha') < Pub(Q', \alpha')$, hence, $Pub(P', \beta) < Pub(Q', \beta)$.
Also by the definition of the $DC$ relation, $\phi$ is followed in $\gamma$ by any mutating event of $Q'$ in $\alpha$ that follows $\phi$ in $\alpha$ and which increases $Q'$'s public pair to be as big as or bigger than $Pub(P', \alpha'\phi)$. It follows from the assumption above, the induction hypothesis and Lemma 4.2.3 that $Pub(Q', \beta) < Pub(P', \beta\phi)$. By the induction hypothesis, Lemma 4.2.3 and because all mutating events of $Q'$ in $\alpha'$ occur in $\beta$, $PubL(Q', \alpha') \leq PubL(Q', \beta)$. We have $PubL(P', \beta\phi) = PubL(P', \alpha'\phi) = PubL(Q', \alpha') \leq PubL(Q', \beta) \leq PubL(P', \beta\phi)$, and so $PubL(P', \beta\phi)$
5. \( \phi \) is a Block event in which a waiting edge is created from process \( P' \) to \( Q' \):

By the definition of the Block transition, \( \text{Pub}(P', \alpha \phi) > \text{Pub}(Q', \alpha') \). Let \( \alpha'' \) be the longest prefix of \( \alpha \) such that \( \text{Pub}(P', \alpha' \phi) > \text{Pub}(Q', \alpha'') \). By Lemma 4.2.3, either \( \alpha'' = \alpha \) or \( \alpha'' \phi \) is a prefix of \( \alpha \) ending in a mutating event of \( Q' \) and \( (\phi, \psi) \in DC(\alpha) \). It follows in either case that \( \text{Pub}(P', \alpha' \phi) > \text{Pub}(Q', \beta) \). Moreover, by the induction hypothesis and Lemma 4.2.4, \( \text{Pub}(P', \alpha' \phi) \) is not a public label of any process in \( \beta \), \( \text{Pub}(P', \alpha' \phi) = \text{Pub}(Q', \beta) \). Since \( \text{Pub}(P', \beta' \phi) = \text{Pub}(P', \alpha' \phi) \), \( \phi \) is enabled after \( \beta \).

**Lemma 4.5** Let \( \alpha \) be a run and \( \pi \) be an event in \( \alpha \). Then \( \text{CC}(\pi, \alpha)(\alpha - \text{CC}(\pi, \alpha)) \) is a run.

**Proof:** By Lemma 4.4, it suffices to show that \( \text{CC}(\pi, \alpha)(\alpha - \text{CC}(\pi, \alpha)) \) is a sequence that is a reordering of \( \alpha \) consistent with \( DC(\alpha) \). Since \( (\phi, \psi) \in DC(\alpha) \) only if \( \phi \) precedes \( \psi^\prime \) in \( \alpha \), \( \text{CC}(\pi, \alpha) \) is finite, and \( \text{CC}(\pi, \alpha)(\alpha - \text{CC}(\pi, \alpha)) \) is a sequence. Suppose \( (\phi, \psi^\prime) \in DC(\alpha) \). Again, \( \phi \) precedes \( \psi^\prime \) in \( \alpha \), so if \( \phi \) and \( \psi^\prime \) are both in \( \text{CC}(\pi, \alpha) \) or both in \( (\alpha - \text{CC}(\pi, \alpha)) \), then \( \phi \) occurs before \( \psi^\prime \) in \( \text{CC}(\pi, \alpha)(\alpha - \text{CC}(\pi, \alpha)) \). Moreover, if \( \psi^\prime \) occurs in \( \text{CC}(\pi, \alpha) \) then \( \phi \) precedes it in \( \text{CC}(\pi, \alpha) \), by the definition of \( \text{CC}(\pi, \alpha) \). Hence \( \phi \) precedes \( \psi^\prime \) in \( \text{CC}(\pi, \alpha)(\alpha - \text{CC}(\pi, \alpha)) \).

**References**


20


