FAULT-TOLERANT DISTRIBUTED DEADLOCK DETECTION/RESOLUTION

Pei-yu Li and Bruce McMillin
CSC-92-04

October 15, 1992

Department of Computer Science
University of Missouri at Rolla
Rolla, Missouri 65401
ff@cs.umr.edu

† This work was supported in part by the National Science Foundation under Grant Number MSS-9216479, and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546.
Abstract

The problem of deadlock detection in a distributed system has been extensively studied in the past few years. Many algorithms on distributed deadlock detection have been proposed under the assumption that the processors and communication in the system are fault-free. However, in an unreliable distributed system, faulty processors may prevent a deadlock detection algorithm from properly detecting deadlocks. Few of the algorithms proposed in the literature address the issue of handling process failures in a distributed system. This paper proposes a fault-tolerant distributed deadlock detection algorithm which integrates a priority-based probe algorithm with a PMC-based diagnosis model. This algorithm detects deadlock cycles as well as identifies process failures under a bounded number of failures in a deadlock cycle by using extended probe messages that contain additional information about faulty processors. We present this algorithm, give an informal proof, and discuss its run time complexity.

Keywords: Fault-tolerant algorithms, distributed deadlock detection, deadlock resolution, distributed algorithms, distributed systems, fault diagnosis.
I. INTRODUCTION

A distributed system consists of a collection of processes which execute on spatially separated processors and communicate via message exchange. A process can request and release resources in an unpredictable order. In such an environment, if resource allocation to processes is not properly controlled, deadlock may occur. Deadlock is a situation in which a set of processes are in a simultaneous wait state, each waiting for one of the others to release its resources before proceeding. The following two definitions present this formally, using a graphical model.

**Definition 1:** A *wait-for graph* (WFG), denoted by WFG(N, E), is a directed graph which depicts the dependencies among processes. N is a finite set of vertices which represent processes in the system. E is a finite set of directed edges where \( i \to j \in E \) if process i is blocked and waits for process j to release a needed resource. If \( i \to j \in E \), we say that process j is the *successor* of process i and process i is the *predecessor* of process j.

**Definition 2:** A *deadlock cycle* is a directed cycle which exists in the WFG.

The problem of deadlock detection in distributed systems has been extensively studied in the past few years. Many previous attempts were made to provide a distributed deadlock detection algorithm for handling resource allocation deadlocks [Ober82, ChMi83, MiMe84, SiNa85, Nata86, BrTo87, RoBu88, ChKo89, KsSi91]. Recently, there has been considerable work on *priority-based probe* algorithms for distributed deadlock detection and resolution [SiNa85, RoBu88, ChKo89, KsSi91]. Generally, in a probe-based algorithm, explicit construction of a centrally maintained WFG is not required, and the presence of a deadlock cycle is verified by propagating special messages called *probes* forward along the edges of the graph (also called an *edge-chasing algorithm*). A probe simply consists of the information about its initiator and some other process. If a deadlock cycle exists in the WFG, eventually the probe will return to its initiator, causing the detection of deadlock. The priority-based algorithm, in addition, assigns a unique
priority to each process. The order of process priorities controls the initiation/propagation of probes, which helps reduce the number of messages initiated for detecting deadlocks.

In a distributed system with low reliability, processor failures are likely to occur. All the probe-based algorithms above are only applicable to a distributed system in which both the communication network and processors are fault-free. In a faulty distributed system, faulty processors may prevent a deadlock detection algorithm from detecting deadlocks. [ElLi85] presented a fault-tolerant deadlock detection algorithm which employs a global detector and local detectors to detect global deadlocks and local deadlocks respectively. However, the algorithm of [ElLi85] is not fully distributed.

In this paper, we propose a fully distributed fault-tolerant deadlock detection algorithm specially designed for distributed systems in which processor failures are likely to occur. The proposed algorithm integrates a priority-based deadlock detection scheme with a fault diagnosis model for the purpose of handling process failures. This deadlock detection scheme is developed by extending the probe-based deadlock detection algorithm of [MiMe84]. The structured system diagnosis model is employed in the WFG for representation of testing, with each vertex of the WFG representing a process/processor. An approach [KuRe80], which ensures that only non-faulty processors communicate information, is applied to the fault location in the proposed algorithm. To make the deadlock detection algorithm fault-tolerant, extended probe messages are incorporated to detect both deadlock and process failure in deadlock cycles. In contrast to probes in most probe-based algorithms [MiMe84, SiNa85, RoBu88, ChKo89, KsSi91], extended probe messages consist of information about faulty processes, in addition to information about probe initiators.

The outline of the paper is as follows: In the next section the fault diagnosis model used in our algorithm is introduced. In Section III the system model is defined. In Section IV we construct our algorithm and discuss its distinguishing features. In Section V an informal proof of the correctness for our algorithm is presented. In Section VI the time complexity of our algorithm is evaluated and we conclude in Section VII.
II. RELATED WORK IN FAULT DIAGNOSIS

The earliest formulated structured system diagnosis model was introduced in [PrMe67]. This model describes a class of systems which can be decomposed into units that are capable of testing other units singly or in combination. In structured system diagnosis, a testing digraph is used, where each vertex represents a process in the system, and an edge $i \to j$ exists only if there is a test in which process $i$ evaluates $j$. It is assumed that non-faulty processes can correctly test all other processes. The test outcome of faulty processes is unreliable and hence a faulty process may determine a non-faulty process to be faulty or a faulty process to be non-faulty. The fault diagnosis algorithm SELF2 in [KuRe80] employs a structured system diagnosis model as its representation of testing. The actual tests involved could be just a simple matter of interrogating the "watchdog" in the processor under test if a processor employs self-testing circuit design techniques and "watchdog" monitoring for error conditions.

Definition 3: A fault vector, $V_i = d_1d_2 \cdots d_n$, in a process $i$ denotes the syndrome of the system from process $i$’s viewpoint. $V_i(j) = d_j = 1$ if process $i$ concludes that process $j$ is faulty and $V_i(j) = d_j = 0$ if process $i$ determines process $j$ to be non-faulty.

In SELF2, each process computes a fault vector containing its conclusions about the conditions of all processes in the system. Initially, a process sets its fault vector to an all zero vector. The process then performs tests on its neighbors periodically and independently of one another, in addition to testing the sender of any diagnostic message. If some faulty process is detected, each process then broadcasts its diagnostic results to all non-faulty processes which tested it. For a process $i$, upon receiving a diagnostic message from a process $j$ that process $i$ determined to be non-faulty by previous testing, process $i$ tests process $j$ again to assure that process $j$ itself is non-faulty. If process $j$ is non-faulty, process $i$ then updates its fault vector with the received diagnostic message and passes the received message on to its non-faulty testers; otherwise, process $i$ sends a message notifying the failure of process $j$ to its non-faulty testers. This process iterates at each process in order to propagate new failure information as quickly as possible and
still keep fault vectors up-to-date. A distributed system of \( n \) processes is \( t \)-fault self-diagnosable if each non-faulty process can correctly identify all faulty and non-faulty processes in the system, provided that no more than \( t \) processes are faulty [KuRe80]. That is, a system employing the above algorithm is \( t \)-fault self-diagnosable if and only if the *connectivity* of its testing digraph is at least \( t \). The connectivity of a strongly connected testing digraph is the minimum number of processes that must be removed to make the digraph not strongly connected.

The above algorithm is based on an approach that a non-faulty process will only accept and pass on diagnostic messages received from the processes which it found to be non-faulty. Since a non-faulty process can correctly test other processes, the non-faulty process will then only accept diagnostic messages from other non-faulty processes. Thus, each non-faulty process eventually has a consistent view of the system, and knows which processes are non-faulty and which are faulty.

### III. FAULT MODEL

Consider a faulty distributed system in which each processor manages the resource allocation and communicates with other processors. A process sends all resource requests to the processor in which it is located. Each processor receives a resource request that can be local or can refer to a resource in another processor, in which case the request is remote. A process can be in two different states, *active* or *blocked* at any time. If a process is blocked, its execution cannot proceed because a requested resource is held by another process, otherwise the process is active. If a processor fails, all processes within it are considered to be faulty as well. If a process is faulty, the entire processor containing this faulty process is also considered to be faulty.

A faulty process is treated as blocked and does not release its resources until the system reconfiguration is made. The processor is the unit of failure and of system reconfiguration. When a processor is configured out of the system, all of its resources are deallocated and made unavailable. All of the faulty units are permanently faulty, by the assumption of the PMC model [PrMe67]. For the sake of simplicity, failures of communication links are assumed to be failures of the incident processors and each processor has
at most one process running on it. A faulty process may invalidate the test results and relay a restricted set of fabricated messages.

**Definition 4:** The size of a deadlock cycle in a WFG\((N, E)\) is the number of vertices involved in the same deadlock cycle.

In this paper, structured system diagnosis is performed using the WFG\((N, E)\) digraph; each blocked process is assumed to be able to test the processes it is waiting on via testing of processors. Formally, if \(i \rightarrow j \in E\), vertex \(i\) is able to test vertex \(j\), and by definition of the WFG, each incident process knows its successor and predecessor. Since we use structured syndrome testing with the PMC model, we are limited to one process failure in each deadlock cycle. Since the size of most deadlock cycles is small [GrHo81], the probability of more than one failure in a cycle is also small (see APPENDIX), hence this is not a problem. The connectivity of a deadlock cycle in the WFG is 1, so the proposed algorithm has \(t = 1\) fault diagnosability. This is consistent with the definition of the fault diagnosis in [PrMe67].

**IV. FAULT-TOLERANT DISTRIBUTED DEADLOCK DETECTION ALGORITHM**

**Definition 5:** A probe is said to be propagated in a *backward* direction if it is sent in the opposite direction along the directed edges of a WFG\((N, E)\). That is, if \(i \rightarrow j \in E\), the probe is sent from \(j\) to \(i\).

[MiMe84] presents an interesting variation of the standard probe-based algorithm, where probes are propagated in a backward direction. The probe propagation is in contrast to the forward propagation of previous probe-based algorithms in Section I. Propagation of label (probe) information in the WFG uniquely locates a process in a cycle which will detect deadlock based on the ordering between the received label and its local label.

In this paper, we adopt and extend the backward priority-based probe approach of [MiMe84] as the underlying scheme because of its elegance and efficiency [KsSi91], plus
suitability for incorporating the fault diagnosis model. When a process is initiated, it is assigned a unique priority such that all processes are totally ordered via a priority function, \( p_r \). This ordering, essentially, can be nothing more than making all process identifiers in the system unique numbers. If fairness is an issue, priorities can be created by a timestamp mechanism [Lamp78]. We say that an antagonistic conflict [SiNa85] occurs if a process of higher priority is waiting for another process of lower priority. That is, in a WFG\((N, E)\), an antagonistic conflict occurs if \( i \rightarrow j \in E \) and \( p_r(i) > p_r(j) \). This priority ordering is used to locate a process to abort as in the following definition.

**Definition 6:** A **deadlock victim** is a process in a deadlock cycle which is selected to abort and to break the cycle.

Once a deadlock victim has aborted itself, all references to its existence must be removed by a clean message which is defined below.

**Definition 7:** A **clean message** is a communicating message which consists of the information being used to clear all probes initiated or propagated by the deadlock victim or faulty process.

Each process \( p \) maintains a probe queue, \( probe\_Q \), and a fault vector, \( V_p \). The probe queue, \( probe\_Q \), contains probes of the form (initiator, successor). Each probe \((q, s)\) indicates that the probe initiated by process \( q \) and received from successor \( s \) faces an antagonistic conflict, i.e., process \( p \) is waiting directly (when \( q = s \)) or indirectly (when \( q \neq s \)) for process \( q \) which has a lower priority. The probe message and clean message are denoted respectively by \( P(D, F) \) and \( C(v, G) \), where \( D \) is a probe set containing the information about probe initiators, \( F \) is a faulty process set showing the faulty processes found recently, \( v \) is the victim that initiates this clean message and \( G \) is a clean message set containing the information being used to clear stale probes in the system. A clean message with \( v = \text{nil} \) implies that a process failure has been identified. Each blocked process can determine whether it is deadlocked by initiating the deadlock detection computation. The process which initiates a deadlock detection computation is called the **initiator** of the computation. Several processes may initiate the deadlock detection computations.
simultaneously and the same process may initiate the computation several times.

The high level model-oriented specification is given in a CSP-like language. In the language, \texttt{do} \texttt{\~} \texttt{od} is a repetition command. A guard command $\square G \rightarrow CL$, is executed only if the execution of its guard, $G$, is true. When a guard is true, $G$ is executed followed by guarded command list, $CL$. A receive command, $q?M$, represents receiving a message $M$ from process $q$. The delay guard becomes true after $timeout$ time period. For a blocked process $p$ and any process $q$,

\begin{verbatim}
P::
do
  if $q$ ? resource_granted() $\rightarrow$ initialization
\square if $q$ ? request_resource() $\rightarrow$ forward_probes
\square if $q$? $P(D, F')$ $\rightarrow$
    [unit_test($q$); check_and_clean($q$);
     [if not in the clean phase $\rightarrow$ collect_and_check_probes($P(D, F')$)
      $\square$ if in the clean phase $\rightarrow$ clean_phase_termination($P(D, F')$)]]
\square if $q$ ? $C(v, G')$ $\rightarrow$ [unit_test($q$); check_and_clean($q$); clean_stale_probes($C(v, G')$)]
\square delay $timeout$ $\rightarrow$
   $[\forall q, (p\rightarrow q)\in E \land V_p(q) = 0 \rightarrow unit_test(q); initiate_probe]$
\od
\end{verbatim}

The detailed algorithm consists of two parts: the \textit{deadlock detection} and \textit{deadlock resolution}. Each part is presented in terms of different cases that may occur. In this algorithm, a "*" means the value in that position is of no concern. For example, (*, s) means all probes in $probe_Q$ with $s$ as the successor, i.e., (*, s) $\equiv$ \{($i, s) \mid (i, s) \in probe_Q$ and $i \in N$\}.

\textbf{A. Deadlock detection}

If process $p$ becomes active because requested resources are granted, procedure $initialization$ is executed.

\textbf{procedure} initialization
empty $probe_Q$, faulty process set $F$, clean message set $G$;
\textbf{if} $p$ is a new process then set fault vector $V_p$ to an all zero vector.

If process $p$ is notified that some process $q$ is now requesting its own resources, procedure $forward_probes$ is executed. Process $p$ sends the detailed information about its $probe_Q$ and fault vector to keep the WFG in a consistent state.
procedure forward_probes
\[ D = \{ i | (i, *) \in \text{probe}_Q \}; \]
\[ F' = \{ i | V_p(i) = 1 \}; \]
sends \( P(D, F') \) to process \( q \).

If process \( p \) receives a probe message from process \( q \), \textit{unit_test} is to verify the status of process \( q \) and ensure that this received message is from a non-faulty process. If process \( q \) is determined to be faulty, process \( p \) will not propagate probes received from process \( q \) by deleting these probes from its \( \text{probe}_Q \) to reduce the number of messages passed around the communication network. The clean phase is an important feature in the deadlock resolution, which is discussed later in the deadlock resolution.

\[
\begin{align*}
\text{procedure } \text{unit_test}(q) \\
\text{if } V_p(q) = 0 \text{ then} \\
\quad \text{test process } q; \\
\quad \text{if process } q \text{ is found to be faulty then} \\
\quad \quad \text{if in the clean phase then} \\
\quad \quad \quad \text{terminate the clean phase} \\
\quad \quad \text{endif}; \\
\quad V_p(q) := 1; \\
\quad F := F \cup \{ q \}; \\
\quad G := \{ i | (i, q) \in \text{probe}_Q \}; \\
\quad \text{delete } (*) , q \text{ from } \text{probe}_Q; \\
\text{endif}
\end{align*}
\]

After \textit{unit_test}(q), if process \( q \) is faulty, process \( p \) initiates a clean message. In addition, if process \( q \) is the predecessor of process \( p \), which is a special case when the deadlock cycle size is 2, since the process failure is not identifiable (see Theorem 4 in Section V), the only solution is to abort process \( p \).

\[
\begin{align*}
\text{procedure } \text{check_and_clean}(q) \\
\text{if } V_p(q) = 1 \text{ then} \\
\quad \text{if } G \neq \emptyset \text{ then} \\
\quad \quad \text{send } C(\text{nil}, G) \text{ to process } j, \text{ where } j \rightarrow p \in E; \\
\quad \quad G := \emptyset \\
\quad \text{endif}; \\
\quad \text{if } q \rightarrow p \in E \text{ then } ll \text{ is the predecessor of } p \text{ ll} \\
\quad \quad G := \{ i | (i, *) \in \text{probe}_Q \}; \\
\quad \quad \text{send } C(p, G) \text{ to process } j, \text{ where } j \rightarrow p \in E; \\
\quad \quad \text{abort} \\
\text{endif}
\end{align*}
\]
If the received probe message shows that one of process $p$’s predecessors has been determined to be faulty, which means a real process failure occurs in a deadlock cycle, process $p$ (the successor of the faulty process) reports the failure to initiate system reconfiguration. The approach of having only the successor identify the process failure has two main advantages: (1) it ensures that only one process reports the process failure in each deadlock cycle, and (2) it helps avoid false deadlock detection. To reduce the load of message communication, process $p$ only propagates probes indicating an antagonistic conflict and deletes other probes from the probe set $D$. If the probe of process $p$ is received, process $p$ declares deadlock and initiates the deadlock resolution by executing $resolution_{_initiation}$. Finally, process $p$ sends out a probe message to its predecessors.

```plaintext
procedure collect_and_check_probes(P(D, F'))
if $V_p(q) = 0$ then
  if $k \in F'$ and $k \rightarrow p \in E$ then
    report the failure of process $k$
  endif
  for all $j \in F'$
    if $V_p(j) = 0$ then
      $V_p(j) := 1$
    else
      $F' := F' - \{j\}$
    endif
  endfor;
  for all $i \in D$
    if $pri(p) > pri(i)$ then
      save $(i, q)$ in $probe_Q$ if $(i, q)$ does not exist
    elseif $pri(p) < pri(i)$ then
      $D := D - \{i\}$
    else
      declare deadlock;
      do $resolution_{_initiation}$
    endif
  endif
endfor;
// assertion: $\forall j \in D, pri(p) > pri(j)$ //
if $D \neq \emptyset$ or $F' \neq \emptyset$ then
  send $P(D, F')$ to process $j$, where $j \rightarrow p \in E$
endif
endif.
```

Deadlock is possible if process $p$ does not receive any message in response to any "?" guard. The delay guard becomes true after $timeout$ seconds and process $p$ tests previously non-faulty successors before initiating a probe message. In $initiate_{_probe}$, when a
process failure is detected, process \( p \) initiates a clean message \( C(\text{nil}, G) \) to eliminate stale probes stored in other processes, restoring the WFG to a consistent state. Finally, process \( p \) sends out a probe message to its predecessors.

\begin{verbatim}
procedure initiate_probe
if \( G \neq \emptyset \) then
    send \( C(\text{nil}, G) \) to the predecessors;
    \( G := \emptyset \)
endif;
send \( P(D=\{ p \}, F) \) to process \( j \), where \( j \rightarrow p \in E; \)
\( F := \emptyset \).
\end{verbatim}

![Figure 1. Example of keeping multiple copies of a probe to help detect future deadlocks.](image)

If a process discards an arriving probe which already exists in its \( \text{probe}_Q \), undetected deadlocks may occur in [SiNa85]. Here, a process keeps multiple copies of a given probe arriving from different successors by labeling them with different successors. In Figure 1, process 3 has the lowest priority in the deadlock cycle \( 3 \rightarrow 6 \rightarrow 7 \rightarrow 3 \). The probe of process 1 is passed through process 3 or process 4 to process 2. Process 2 then keeps two copies of the probe of process 1. When process 3 detects deadlock, it will send a clean message \( C(3, \{ 1 \}) \) to process 2. Process 2 deletes probe \( (1, 3) \) and keeps \( (1, 4) \). Therefore, even if process 1 becomes active and waits for process 2 (denoted by the dashed edge), forming a new deadlock cycle \( 1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \), this new deadlock will still be detected when process 1 receives a probe message \( P(\{ 1 \}, \{ \}) \) from process 2.
Once a processor (process) failure is detected, each processor which receives the failure report releases its resources held by these faulty processes. Future messages from the faulty processor are ignored by any non-faulty processor, thus, performing reconfiguration by isolating the faulty processor.

B. Deadlock resolution

When the deadlock is detected, the detector becomes the deadlock victim and initiates deadlock resolution by entering the clean phase. Since the victim aborts itself, it is necessary to clear all probes in each probe_Q which were initiated or propagated by the victim by initiating a clean message C(v, G). While a process is in the clean phase, it will not send out or relay any message to other processes until the clean phase has terminated.

If process p receives its probe back and becomes a deadlock victim, procedure resolution_initiation is executed.

\[\text{procedure resolution_initiation} \]
\[G = \{i | (i, *) \in \text{probe}_Q\};\]
sends C(p, G) to process j, where j→p ∈ E.

If process p receives a clean message from process q, process p tests for failure and then clear old probe information. In clean_stale_probes, process p deletes stale probes from its probe_Q according to the received message, then passes this message to its predecessors. If process p is already in the clean phase and receives its clean message back (when v = p), then it aborts.

\[\text{procedure clean_stale_probes}(C(v, G'))\]
\[\text{if } V_p(q) = 0 \text{ then}\]
\[\text{if in the clean phase then}\]
\[\text{if } v = p \text{ then}\]
\[\text{abort}\]
\[\text{endif}\]
\[\text{endif}\]
\[\text{else}\]
\[\text{delete } (v, q) \text{ from probe}_Q;\]
\[\text{delete } (i, q) \text{ from probe}_Q, \text{ where } i \in G';\]
\[\text{send } C(v, G') \text{ to process } j, \text{ where } j→p \in E.\]
\[\text{endif}\]
\[\text{endif}\]
If process \( p \) is in the clean phase and receives a probe message from process \( q \), process \( p \) tests for failure, and terminates the clean phase if a failure message is received. Similar to [KsSi91], deadlock victim \( p \) does not abort until it receives its own clean message. Unlike [KsSi91], a dynamic clean phase is employed where process \( p \) is allowed to terminate its clean phase when it receives a failure message.

```
procedure clean_phase_termination(P(D, F'))
if \( V_p(q) = 0 \) then
  if \( F' \neq \emptyset \) then
    terminate the clean phase;
    do collect_and_check_probes(P(D, F'))
  endif
endif
endif.
```

This, in turn, helps solve problems resulting from process failures in the deadlock resolution; otherwise a faulty process may cause the deadlock victim to be blocked indefinitely. Consider the deadlock cycle shown in Figure 2. Assume process 1 detects deadlock and sends out a clean message. Process 3 fails prior to receiving the clean message, preventing process 1 from receiving its clean message back. Thus, this deadlock may remain forever. Here, process 4 will diagnose the failure of process 3 and send out a probe message with this failure message. Once process 1 receives this failure message, it will terminate its clean phase and propagate this failure message to process 2. Finally, process 2 will report the process failure to resolve deadlock.

Figure 3 shows a simple example explaining how this algorithm works. The order of priority among processes in the WFG is 5 > 4 > 3 > 2 > 1. (a) Initially, a deadlock

![Figure 2. Example of solving problems resulting from the process failure in the deadlock resolution.](image-url)
cycle $2 \rightarrow 4 \rightarrow 3 \rightarrow 2$ exists in the WFG. Processes 2, 3, and 4 save the probe of process 1, which has the lowest priority in the WFG. Process 4 also saves the probe of process 3. (b) After waiting for a predefined period of timeout period, process 2 detects the failure of process 5 and sends out $P(2, 5)$ to process 3. Processes 3 and 4 save the probe of process 2 and update their own fault vector. (c) Process 2 receives its own probe, enters the clean phase, and sends out $C(2, 1)$ to process 3. Processes 3 and 4 then eliminate the
probes associated with process 1 and 2.

V. CORRECTNESS OF THE ALGORITHM

A deadlock detection algorithm is correct if it detects every deadlock that occurs and does not detect any false deadlock. In this section, we will show that the proposed algorithm is correct under the following assumptions:

1) No multiple deadlock victims concurrently exist in a deadlock cycle, i.e., no non-victim process in a deadlock cycle is involved in another cycle at the same time; otherwise, there is no way to prevent false deadlock detection, since one deadlock victim in the cycle could have aborted when some other victim declares deadlock.

2) Any process in a deadlock cycle does not abort until the deadlock is detected and the resolution is initiated.

3) There exists at most one process failure in a deadlock cycle (the probability of more than one is very small [LiMc92]).

Lemma 1: If a deadlock occurs, there exists a cycle in the WFG.
Proof: This follows from four necessary conditions for a deadlock in [CoEl71]. □

Lemma 2: In the absence of processor failure, only the probes with a priority lower than or equal to the lowest one in the deadlock cycle propagates around this cycle.
Proof: A process sends out a probe message when it either times out or receives a probe which faces an antagonistic conflict. By the assertion, \( \forall j \in D, \text{pri}(p) > \text{pri}(j) \), in procedure collect_and_check_probes of the algorithm, a process will not propagate any received probe with a priority higher than its own. It is obvious that in the absence of processor failure, no probe with a higher priority will be able to pass through the process with the lowest priority in the deadlock cycle and return to its initiator. □

Theorem 1: If a deadlock occurs, in the absence of processor failure, it is detected by the process of the lowest priority in this deadlock cycle.
Proof: By Lemmas 1 and 2, when a deadlock occurs, only the probe with the lowest priority in the deadlock cycle can pass around this cycle and return to its initiator. Thus, only the process of the lowest priority in this cycle declares the deadlock. □

![Proof diagram](image)

Figure 4. Process 1 has the lowest priority in the cycle and initiates the probe \( P \). The dashed lines indicate the direction that \( P \) traverses. (a) \( P \) passes the faulty process 3 before the failure occurs, and returns to its initiator process 1 (the deadlock is detected). (b) \( P \) passes the faulty process 3 after the failure occurs, but is discarded by process 4 (no deadlock is detected).

**Theorem 2:** This algorithm detects deadlock only if there exists a deadlock.

Proof: This algorithm detects deadlock when the process of the lowest priority in a deadlock cycle receives its own probe. If there exists a process failure in this cycle, it must happen after this probe has passed the faulty process. That is, this probe must reach its initiator before the failure message. Therefore, it is impossible that a faulty process is aborted before the deadlock is detected (see Figure 4). Since the deadlock detector is the only victim in the deadlock cycle and no non-victim process will abort itself, there must exist a real cycle of waiting processes. □

**Theorem 3:** In a deadlock cycle of size greater than two, the failure of a process, \( i \), is only identified by its successor, \( j \), if \( i \rightarrow j \) is part of the cycle.

Proof: In a deadlock cycle with \( k \rightarrow i \rightarrow j \) as part of the cycle, when process \( i \) fails, the predecessor, \( k \), of process \( i \) will diagnose this failure and send out a probe message with the failure message in a backward direction in the cycle. Since the cycle size is greater than two and only one process fails, there must be at least one non-faulty process (including the successor, \( j \), of process \( i \)) between process \( k \) and process \( i \) in the path through which
the failure message traverses. It is guaranteed that the failure message eventually reaches process \( j \). Since \( i \rightarrow j \in E \), as shown in procedure \texttt{collect\_and\_check\_probes} of the algorithm, only this successor \( j \) of faulty process \( i \) identifies the process failure in the deadlock cycle. \( \square \)

Figure 5. Examples of process failures which are not identifiable. (a) Faulty process \( j \) determines process \( i \) to be non-faulty and relays the probe received from process \( i \). Process \( i \) discards the probe since process \( j \) is determined to be faulty. (b) Faulty process \( j \) determines process \( i \) to be faulty and discards the probe received from process \( i \).

**Theorem 4:** In a deadlock cycle of size two, the process failure is not identifiable.

**Proof:** The reason for this comes from the theory of structured system diagnosis [PrMe67], we need \( 2t+1 \) processors for a one-step \( t \)-fault diagnosable system. In a deadlock cycle consisting of a non-faulty process, \( i \), and a faulty process, \( j \), either one is the successor and the predecessor of the other. Process \( i \)'s test on process \( j \) will indicate that process \( j \) is faulty. As mentioned earlier, a faulty process may invalidate its testing results. Process \( j \)'s test on process \( i \) may also show that process \( i \) is faulty. In which case, neither of them accepts the messages from the other. Consequently, process \( i \) will either discard or not receive the probe message showing the failure of process \( j \), having no process failure identified. Figure 5 illustrates above two cases. \( \square \)

**VI. COMPLEXITY**

In this section, we analyze the algorithm’s time complexity. Since transmitting a message through the communication network is usually a time-consuming work, we adopt this cost as our time unit and evaluate the algorithm’s complexity by analyzing the number of probe messages generated between the occurrence and detection of a deadlock.
cycle with size $n$. Since there are no natural models that capture the general behavior of our algorithm, we count the number of probe messages generated in the best and worst configurations of a deadlock cycle. We will not count the number of clean messages since this number is not greater than the number of probe messages.

**Theorem A.1**: The average lower bound on the expected number of probe messages is $O(n)$.

*Proof*: See APPENDIX.

**Theorem A.2**: The average upper bound on the expected number of probe messages is $O(n^2)$.

*Proof*: See APPENDIX.

**Theorem A.3**: The average worst-case communication cost is $O(Tn^2)$ with the process timeout period of $T$.

*Proof*: See APPENDIX.

**VII. CONCLUSION**

We have presented a fault-tolerant distributed algorithm to detect and resolve deadlock in a distributed system in which there is at most one process failure in each deadlock cycle. The work in this paper is motivated by the desire to cope with problems resulting from process failures, which were ignored by most existing algorithms. A novel feature of the proposed algorithm is that it integrates a priority-based probe scheme with a PMC-based diagnosis model, and uses extended probe messages to detect both deadlock and process failure in deadlock cycles, significantly lessening the message traffic in the communication network. The communication cost is evaluated by counting the number of probe messages generated in the best and worst configurations of a deadlock cycle, which respectively are $O(n)$ and $O(n^2)$. 
We cannot extend this result to more than one process failure in each deadlock cycle as the result of [PrMe67] shows. However, since the size of most deadlock cycles is small [GrHo81], the probability of more than one failure in a cycle is also small, hence this limitation is not a problem.

**APPENDIX**

In this appendix we first calculate the probability of more than one process failure in a deadlock cycle and then analyze the communication cost between deadlock occurrence and deadlock detection by counting the number of probe messages generated in two kinds of configurations of a deadlock cycle [SiNa85]. The priority order of processes in the deadlock cycle is \( \text{pri}(i) < \text{pri}(j) \) if \( i < j \). It is assumed that each process is equally likely to fail, with a failure rate \( p \). Since a process failure in a deadlock cycle is a special case which causes deadlock, identification of process failure is viewed as detection of deadlock.

Given a deadlock cycle of size \( n \), the probability of more than one process failure in the cycle is

\[
Pr[\# \text{ of process failures in a deadlock cycle} > 1] = 1 - Pr[\# \text{ of process failures in a deadlock cycle} \leq 1] = 1 - (1 - p)^n - np(1 - p)^{n-1}.
\]

Let \( n \leq 5 \) and \( p = 0.1 \), then \( Pr[\# \text{ of process failures in a deadlock cycle} > 1] \leq 0.08146 \).

Given the condition that a deadlock of cycle size \( n \) has occurred and at most one process failure is allowed in this cycle, the probability that a deadlock is detected in a fault-free deadlock cycle is

\[
Pr[\text{fault-free cycle} | \text{deadlock has occurred}] = \frac{(1 - p)^n}{(1 - p)^n + np(1 - p)^{n-1}} = \frac{1 - p}{pn + 1 - p}.
\]

The number of probe messages generated in the fault-free deadlock cycle and faulty
deadlock cycle are denoted by $N_{\text{fault-free}}$ and $N_{\text{faulty}}$, respectively. Hence, the expected number of probe messages generated between deadlock occurrence and deadlock detection is

$$E = Pr[\text{fault-free cycle} \mid \text{deadlock has occurred}] N_{\text{fault-free}} + (1-Pr[\text{fault-free cycle} \mid \text{deadlock has occurred}]) N_{\text{faulty}}.$$ 

![Figure 6](image)

**Theorem A.1**: The average lower bound on the expected number of probe messages is $O(n)$.

**Proof**:  

The best deadlock configuration is the one in which only one edge of the deadlock cycle causes an antagonistic conflict. Consider the configuration illustrated in Figure 6(a), only edge $n \rightarrow 1$ causes an antagonistic conflict, that is, only the probe initiated by process 1 can pass its predecessor.

When the deadlock cycle is fault-free, the probe of process 1 traverses around the cycle and returns to process 1 before the other processes time out. In this case, the number of messages generated for deadlock detection is $N_{\text{fault-free}} = n$. When some process, say process 2, fails and its predecessor (process 1) detects the failure right after the deadlock cycle is formed, the probe of process 1 goes through the cycle to process 3 and terminates. In this case, $N_{\text{faulty}} = n - 2$. Hence, the expected number of probe messages
generated in the best configuration is

\[ E_{\text{BEST}} = Pr[\text{fault-free cycle } | \text{ deadlock has occurred}] \ N_{\text{fault-free}} + (1-Pr[\text{fault-free cycle } | \text{ deadlock has occurred}]) \ N_{\text{faulty}} = \frac{pn^2 + (1-3p)n}{pn + 1 - p} = O(n). \]

**Theorem A.2:** The average upper bound on the expected number of probe messages is \( O(n^2) \).

**Proof:**

The worst case is the one in which each edge of the cycle except one causes an antagonistic conflict. Figure 6(b) shows an example of the worst deadlock configuration in which only edge 1\( \rightarrow \)\( n \) does cause an antagonistic conflict. Each probe message initiated in this configuration traverses around the deadlock cycle until process 1 or some faulty process is reached.

When the deadlock cycle is fault-free, the probe messages of all processes traverse to process 1 and terminate. In this case,

\[ N_{\text{fault-free}} = 1+2+...+n = \frac{n(n+1)}{2}. \]

The analysis of communication cost in a faulty deadlock cycle is more complex. We have to evaluate the cost in each case when different process failures occur.

- **When process 1 fails before its probe message returns:**

  The probe messages of all processes traverse to process 1. Process 2 detects failure after the other processes time out and send out their probe messages. In this case, the number of messages generated is
\[ 1 + 2 + \ldots + (n-1) + n = \frac{n(n-1)}{2} + n \] 
\[ + [1 + 2 + \ldots + (n-2) + (n-2)]. \]

where the first term is the cost before process failure and the second term is the cost after process failure.

* When process \( n \) fails before the probe message of process 1 passes it:

  The probe messages of all processes except process 1 traverse to process 1 and the probe message of process 1 stops at process \( n \). Process 1 detects failure after the other processes time out. In this case, the number of probe messages generated is

\[ [1 + 2 + \ldots + (n-1) + (n-1) = \frac{n(n-1)}{2} + (n-1)] + [1 + 2 + \ldots + (n-2) + (n-2)], \]

where the first term is the cost before process failure and the second term is the cost after process failure.

Likewise, we evaluate the communication cost in other cases as follows:

* process \( n-1 \) fails: \[ \frac{n(n-1)}{2} + (n-2) \] 
  \[ + [1 + 2 + \ldots + (n-2) + (n-2)] \]

* process \( n-2 \) fails: \[ \frac{n(n-1)}{2} + (n-3) \] 
  \[ + [1 + 2 + \ldots + (n-3) + 1 + (n-2)] \]

* process \( n-3 \) fails: \[ \frac{n(n-1)}{2} + (n-4) \] 
  \[ + [1 + 2 + \ldots + (n-4) + 1 + 2 + (n-2)] \]

  .
  .
  .

* process 3 fails: \[ \frac{n(n-1)}{2} + 2 \] 
  \[ + [1 + 2 + 1 + 2 + \ldots + (n-4) + (n-2)] \]

* process 2 fails: \[ \frac{n(n-1)}{2} + 1 \] 
  \[ + [1 + 1 + 2 + \ldots + (n-3) + (n-2)] \]
\[ N_{\text{faulty}} = \frac{5n^3 + 3n^2 - 14n + 6}{6} \]

Hence, the expected number of probe messages generated in the worst configuration is

\[
E_{\text{WORST}} = Pr[\text{fault-free cycle | deadlock has occurred}] \cdot N_{\text{fault-free}} + \\
(1 - Pr[\text{fault-free cycle | deadlock has occurred}]) \cdot N_{\text{faulty}}
\]

\[ = \frac{5pn^3 + 3n^2 + (3 - 17p)n + 6p}{6(pn + 1 - p)} \]

\[ = O(n^2). \qed \]

**Theorem A.3**: The average worst-case communication cost is \( O(Tn^2) \) with the process timeout period of \( T \).

**Proof**:

If we assume that each process joins a deadlock cycle with an interarrival time obeys the exponential distribution, and the timeout period of each process is fixed, we can model the average case in the worst deadlock configuration. Consider that we have \( n \) independent processors in a deadlock cycle with the same interarrival rate \( \lambda \). The probability of \( x \) timeouts (or process arrivals) in a time interval of length \( T \) is Poisson:

\[ P_x(T) = \frac{(n\lambda T)^x}{x!} e^{-n\lambda T}, \]

and then the expected number of timeouts occur in a time \( T \) is

\[ E_{\text{timeout}}(T) = n\lambda T. \]

Since each process is independent, for the simplicity of analysis in both fault-free and faulty cases, we assume that all probe messages originate at halfway point in the deadlock cycle. Thus, the number of probe messages generated for each timeout is \( \frac{n}{2} \).
The timeout period, $T$, is chosen as the maximum time required for the probe message to get to the deadlock detector or faulty process.

When the deadlock cycle is fault-free, $n\lambda T$ timeouts occur before the probe of process 1 returns to process 1 (the deadlock detector). In this case, $N_{\text{fault-free}} = n\lambda T \times \frac{n}{2} = \frac{n^2 \lambda T}{2}$. When some process fails in the deadlock cycle, $n\lambda T$ timeouts occur before the probe of process 1 reaches the faulty process and $n\lambda T$ timeouts appear between the occurrence and the identification of process failure. Thus, in this case, $N_{\text{faulty}} = 2n\lambda T \times \frac{n}{2} = n^2 \lambda T$. The average communication cost in the worst configuration is then

$$E_{\text{Worst}} = Pr[\text{fault-free cycle} \mid \text{deadlock has occurred}] \cdot N_{\text{fault-free}} + (1-Pr[\text{fault-free cycle} \mid \text{deadlock has occurred}]) \cdot N_{\text{faulty}}$$

$$= \frac{2p\lambda Tn^3 + (1-p)\lambda Tn^2}{2(pn+1-p)}$$

$$= O(Tn^2). \square$$

REFERENCES


