Determining the Last Process to Fail

DALE SKEEN
Cornell University

A *total failure* occurs whenever all processes cooperatively executing a distributed task fail before the task completes. A frequent prerequisite for recovery from a total failure is identification of the last set (LAST) of processes to fail. Necessary and sufficient conditions are derived here for computing LAST from the local failure data of recovered processes. These conditions are then translated into procedures for deciding LAST membership, using either complete or incomplete failure data. The choice of failure data is itself dictated by two requirements: (1) it can be cheaply maintained, and (2) it must afford maximum fault-tolerance in the sense that the expected number of recoveries required for identifying LAST is minimized.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4: [**Performance of Systems**]: *reliability, availability, and serviceability*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault-tolerance*; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*

General Terms: Algorithms, Reliability

Additional Keywords and Phrases: Cooperative processes, total failure, event ordering

1. INTRODUCTION

A *total failure* occurs whenever all processes cooperatively executing a distributed task fail before the task completes. Frequently, a prerequisite for recovering from a total failure is reconstructing the task state immediately prior to the total failure. This, in turn, requires identification of the last process to fail. Since processes can fail concurrently, as well as sequentially, the general problem is to identify the set, denoted LAST, of processes failing last.

This problem arises in several contexts in highly fault-tolerant distributed systems. In a distributed database system, a transaction is typically managed by a group of *transaction coordinators*, whose principal responsibility is to decide whether to commit or abort the transaction. Normally a designated coordinator, the primary, actually decides, while the others serve as backups [2]. If the primary fails before deciding, one of the backups is promoted into the decision-making role. A total failure in this context occurs when all coordinators fail. When such a failure occurs, the last coordinator to fail must be identified in order to determine if a decision had been made (and, if so, to complete the transaction accordingly).

Author's address: IBM Research Laboratory, San Jose, CA 95193.

A second application of the problem of computing LAST arises in the management of replicated data. Replicated data is usually managed by a group of processes, known as *data managers*, each of which controls access to one copy. One popular data access algorithm allows the replicated data to be modified whenever any manager is operational, and forwards each modification to all currently operational managers [1]. In order to recover from the failure of all data managers, the most up-to-date copy of the data must be determined. This, of course, requires identification of the last data manager to fail.

In this paper we derive necessary and sufficient conditions for computing LAST from the "available information." We assume that each process maintains some local failure information on stable storage, which is available whenever the processor containing that information is operational. We desire that the failure data be maintainable with little overhead, yet provide a high degree of fault-tolerance in the sense that the expected number of recovered processors required for identifying LAST is small. Two factors complicate the problem: in many situations failure information is incomplete, and the set of processes executing the task is often dynamic—processes are continuously added to and deleted from the set.

The remainder of this paper is organized as follows. The next section defines the processing environment and the failure information maintained by each process. Basic results on reconstructing the failure ordering from the stored failure data are also given. Section 3 shows how to infer LAST when each process knows the identities of all other processes executing the assigned task. Section 4 extends these results to systems in which each process knows only a subset of its fellow members. Section 5 discusses implementation issues, including how to prune failure data for long-lived tasks.

2. BACKGROUND

2.1 The Environment

We assume a distributed system of asynchronous processes, communicating solely through messages. Each process has a single lifetime: after creation, it is *operational* until eventually it fails. Processes are assumed to fail by halting (crash failures), an assumption that is used in the design of many distributed systems. Note that if processes could fail by skipping steps in their program (omission failures) or by acting arbitrarily (Byzantine failures), then identification of LAST would not suffice to reconstruct a meaningful task state. The message delivery system is assumed to be completely reliable: messages between operational processes are never lost and are delivered within finite but arbitrary time. However, messages may be delivered in *any* order.

A failure is detectable by any process attempting to communicate with the failed process. So that all information, including failure information, is conveyed through messages in a uniform manner, we assume that a failing process sends *failed*(j) messages to all processes. Such messages are delivered and processed in the same way that normal messages are. In practice, of course, failures would be detected by using timeouts or, if more reliability is desired, by a failure detection protocol such as the one described in [4].

A *process group* is a finite, nonempty set of communicating and cooperating processes. If the membership of the group is fixed over time, the group is said to be *static*; otherwise, it is said to be *dynamic*. A new process can *join* a dynamic group by executing a suitable protocol. Among other things, the protocol makes the new member known to the other members of the group. For convenience, we assume that a member can never quit a group (i.e., become a nonmember). Although a process may join several groups, we will consider its participation in each group separately. Throughout this paper, P denotes the process group of interest. All definitions and results are relative to P.

Upon joining a group, a process becomes an *operational member* and remains so until it fails, whereupon it is classified a *failed member*. The evolution of each process with respect to a process group is thus

$$\text{nonmember} \longrightarrow \text{operational member} \longrightarrow \text{failed member.}$$

A *total failure* within a process group occurs when all members have failed.

2.2 Event Ordering

In contrast to a centralized system, the temporal ordering of events in a distributed system is only partial. Accordingly, understanding a distributed system requires some knowledge of partial orderings. The terminology and properties of partial orderings used in this paper are given in the appendix.

We are interested in ordering *events* in a way that is consistent with causality. For an event at process i to affect an event at process j, communication must occur between the two processes. In a message-based system, this communication must consist of messages, either a single message sent by i and received by j, or a sequence of messages where i sends a message to k, k sends to $l, \ldots,$ and m sends to j. Therefore, the event at process i should be ordered before the event at process j only if such a sequence of messages exists. In [3] a partial ordering, the "happens before" relation with this property, is defined as follows.

Definition [3]. The *happens before* relation, denoted by $\longrightarrow$, on a set of events is the smallest relation satisfying the following three conditions:

(1) If a and b are events in the *same* process, and a precedes b according to some local clock, then $a \longrightarrow b$.
(2) If event a is the sending of a message and event b is the receipt of the same message, then $a \longrightarrow b$.
(3) If $a \longrightarrow b$ and $b \longrightarrow c$, then $a \longrightarrow c$.

Distinct events a and b are said to be *concurrent* if neither $a \longrightarrow b$ nor $b \longrightarrow a$.

Clearly, event a can affect event b only if $a \longrightarrow b$. Concurrent events, therefore, cannot affect one another. Events of interest to us include the sending and the receiving of messages, the joining of a process to a process group, and, most important, process failures and their detection. Hereinafter, all references to event orderings are understood to be references to the "happens before" relation, as defined above.

The appeal of our particular failure model is that failures and failure messages need not be treated specially in the above definition. The failure of process i can affect events at another process j only if there is communication between i and j after the failure occurs. As always, this communication is in the form of a sequence of messages, but in this case the first message in this sequence must be failed(i), since failure messages are the only type of messages that can be sent by a failed process. This implies the useful and intuitive observation that i's failure can affect events in j only if some process (not necessarily j) receives failed(i). (That is to say, i's failure can affect events at process j only if i's failure is detected by some process k. Note that j and k may be different processes.)

The *failed before* relation, denoted FB, is derived from the subrelation of "happens before" concerned with failures. For convenience, we define it over processes rather than events and say *i failed before j* if and only if i's failure could have affected some event in j.

Definition. Let i and j be two processes. i FB j if and only if there exists some event e_j in j such that (i's failure) $\rightarrow e_j$.

Two processes, i and j, fail *concurrently* if neither i FB j nor j FB i. That FB is a partial ordering follows from the definition of $\rightarrow$.

We are now ready to define LAST. Intuitively, LAST is the set of processes whose failure could not have affected events at other processes. Formally,

Definition. LAST $= \{j \mid \neg \exists i (j \text{ FB } i)\}$.

Equivalently, LAST is the set of maximal elements[1] of P with respect to the partial ordering FB.

Using the above definitions, it is a straightforward exercise to show that the definition of LAST captures the intended semantics: $i \in$ LAST *if and only if i's failure did not affect any event at another process.* From this, it follows that $i \in$ LAST if and only if no process in P received the message failed(i).

2.3. Failure Information

In order to determine LAST after a total failure, the processes in a group maintain failure information on nonvolatile storage about their fellow group members. These data are readable by recovery processes after a total failure has occurred. Specifically, each process i in group P maintains two sets:

(1) P_i—*i's cohort set*—a subset of P known to i.
(2) f_i—*i's mourned set*—a subset of P_i composed of processes that have failed and whose failures are known to i. (i "mourns" the members of f_i.)

Although the values of P_i and f_i vary over time, we are interested in them only after i has failed, at which time their values are fixed.

A process's cohort set is *complete* if it equals P and is *incomplete* otherwise. Similarly, a process's mourned set is *complete* if it contains all processes that have failed before it and is *incomplete* otherwise. Incomplete cohort sets can arise when process groups are dynamic, because a process will normally not know the processes joining after its failure. Incomplete mourned sets are sometimes intentionally maintained because of cost considerations. As we discuss in Section 5,

[1] For a definition of maximal element, see the appendix.

the maintenance of complete mourned sets is itself a nontrivial and sometimes expensive task.

If LAST is to be determinable from the failure data, a certain minimum amount of information must be maintained. Process i's cohort set must contain i and f_i. Its mourned set must record each failure that it detected directly through the receipt of a failure message. These constraints ensure that

PROPOSITION 1. $\bigcup_{i \in P} P_i - \bigcup_{i \in P} f_i = LAST$.

PROOF. Because of the requirement $i \in P_i$, we have $\bigcup_{i \in P} P_i = P$. Consequently, we need to show only that $i \notin \bigcup_{j \in P} f_j$ if and only if $i \in$ LAST. This follows from (1) $i \in$ LAST if and only if no process received a failed(i), and (2) $i \in f_j$, for some j, if and only if some process received failed(i). □

After a total failure, it is often the case that only failure data from a proper subset of the processes in the group are available for reading by recovery processes (the remaining failure data being unavailable because e.g., the processors holding the data are not operational). It is from the collection of available cohort sets and mourned sets that LAST is to be determined. An initial, nonempty collection determines a set of candidates for LAST. As time progresses and more processors recover, more data become available, and the set of candidates shrinks. Our primary task is to determine when sufficient data are available to conclude that the candidate set equals LAST. A simpler task, which is satisfactory for many applications, is to identify a single member of LAST.

The available failure data induce a partial ordering among process failures. Even when all data are available, this ordering is generally weaker than the failed-before relation because of the possible incompleteness of mourned sets. Nonetheless, this induced ordering will be instrumental in proving the correctness of algorithms for calculating LAST. This ordering is denoted by FB_R, where R is the set of processes with accessible failure data.

Definition. Let $R \subseteq P$. FB_R is the smallest relation on processes in P satisfying

(1) if $j \in R$ and $i \in f_j$, then $i\ \mathrm{FB}_R\ j$, and
(2) if $i\ \mathrm{FB}_R\ j$ and $j\ \mathrm{FB}_R\ k$, then $i\ \mathrm{FB}_R\ k$.

If all f_i's are complete, then FB_R is said to be complete.

FB_R is the irreflexive transitive closure of the information contained in the failure data of processes in R. Notice that it defines a partial ordering over all processes in P, not just the ones in R.

We can view FB_R as defining a set of possible candidate relations for the unknown FB relation. Each candidate relation is called a *feasible extension of* FB_R and is characterized as follows.

Definition. A binary relation *fb* over processes is called a *feasible extension* of FB_R if and only if

(1) fb is a partial ordering, and
(2) fb extends FB_R(i.e., $\mathrm{FB}_R \subseteq \mathrm{fb}$).

In addition, if FB_R is complete, then fb must also satisfy

(3) If i fb j and $j \in R$, then i FB_R j.

Each feasible extension is consistent with the available failure data and hence can not be excluded from consideration.

3. DETERMINING LAST IN A STATIC GROUP

In this section we assume that process group P is fixed and known to every member process; hence, cohort sets are complete. This is typical of static groups, where group membership remains invariant over time. With this assumption, we derive necessary and sufficient conditions for determining LAST. As before, R denotes the subset of processes whose failure data is available.

The cases of complete and incomplete mourned sets are considered separately. We assume that the completeness of the failure data is known a priori rather than inferred from the data itself. We show toward the end of this section that testing completeness is as difficult as testing LAST membership.

To simplify the notation, we define

$$\mathrm{CAND}_S \equiv P - \textstyle\bigcup_{i \in S} f_i, \qquad \text{where} \quad S \subseteq P.$$

CAND_S is the set of processes that failed before no other process according to the mourned sets of the processes in S; consequently, for any S, we have

$$\mathrm{LAST} \subseteq \mathrm{CAND}_S. \tag{3.1}$$

Think of CAND_S as being the candidates for LAST membership according to processes in S. By Proposition 1 it follows that $\mathrm{LAST} = \mathrm{CAND}_S$ whenever $S = P$. We will be interested primarily in the set CAND_R, the members of which are the candidates for LAST according to the available failure data.

3.1 Using Complete Information

With complete mourned sets the key to an efficient LAST membership test is the observation that the members of LAST record sufficient information to determine LAST.

LEMMA 1. *With complete mourned sets, $CAND_{LAST} = LAST$.*

PROOF. Since $\mathrm{LAST} \subseteq \mathrm{CAND}_{\mathrm{LAST}}$ (by 3.1), we need only show that $\mathrm{CAND}_{\mathrm{LAST}} \subseteq \mathrm{LAST}$ to establish the lemma.

Let i be an arbitrary member of $\mathrm{CAND}_{\mathrm{LAST}}$. Since mourned sets are complete, i's membership in $\mathrm{CAND}_{\mathrm{LAST}}$ implies that i failed before no member of LAST. Now, a process fails before no member of LAST if and only if it is a member of LAST (by Theorem A2 in the appendix). Therefore, $i \in \mathrm{LAST}$, and this establishes that $\mathrm{CAND}_{\mathrm{LAST}} \subseteq \mathrm{LAST}$. □

With complete mourned sets, LAST is determinable if mourned sets from all its members are available. This is useful only if there is an effective way to check whether the data for all members of LAST is available without explicitly knowing LAST. Fortunately, there is a simple check: If the candidates for LAST constitute a subset of R, then by Lemma 1, LAST is a subset of R. The next theorem uses this check in determining LAST.

```
procedure DeterminiLast(i, P, $f_i$);
local j: process;
      R, $\text{CAND}_R$, $f'$: set-of-processes;
begin
  $R := \{i\}$;
  $\text{CAND}_R := P - f_i$;
  broadcast message $(i, f_i)$ to P;
  while $(\text{CAND}_R \not\subseteq R)$ do begin
    $(j, f') :=$ receive( );
    $R := R \cup \{j\}$;
    $\text{CAND}_R := \text{CAND}_R - f'$;
    end;
  print $\text{CAND}_R$;
end;
```

Fig. 1. The distributed algorithm for determining LAST given complete cohorts and mourned sets. The above procedure is executed by the recovery process for process i, for all $i \in R$.

THEOREM 1. *Let R be an arbitrary subset of P, and assume the completeness of mourned sets. If $CAND_R \subseteq R$, then $CAND_R = LAST$.*

PROOF. Again, since we know LAST $\subseteq \text{CAND}_R$ (by 3.1), we need only establish

$$\text{if} \quad \text{CAND}_R \subseteq R, \qquad \text{then} \quad \text{CAND}_R \subseteq \text{LAST}.$$

Assume $\text{CAND}_R \subseteq R$. First, let us note that this assumption and the fact that LAST $\subseteq \text{CAND}_R$ gives us (by transitivity)

$$\text{LAST} \subseteq R. \tag{3.2}$$

Now, let us prove $\text{CAND}_R \subseteq$ LAST, which, by Lemma 1, is equivalent to

$$\text{CAND}_R \subseteq \text{CAND}_{\text{LAST}}.$$

Replacing CAND with its definition gives us

$$P - \bigcup_{i \in R} f_i \subseteq P - \bigcup_{i \in \text{LAST}} f_i,$$

Eliminating the common term and rearranging gives us

$$\bigcup_{i \in \text{LAST}} f_i \subseteq \bigcup_{i \in R} f_i. \tag{3.3}$$

The validity of (3.3) follows from (3.2). □

An algorithm for determining LAST, based on Theorem 1, is given in Figure 1. For each process i in R, an associated recovery process executes the given procedure. The recovery processes exchange failure data and incrementally evaluate, in parallel, the expression $\text{CAND}_R \subseteq R$. (Note that variable R is local to each process, and its value may vary from process to process, depending on the message delivery order.) A recovery process terminates when $\text{CAND}_R \subseteq R$ evaluates to true and, as a result, LAST is determined.

Theorem 1 establishes that the availability of failure data from all candidates for LAST suffices to determine LAST. The next theorem establishes that this information is also necessary: If $\text{CAND}_R \not\subseteq$ LAST, then, in all cases, LAST membership for at least one process is indeterminable.

The observed ordering of failures:

1	failed before	2
2	failed concurrently with	3
2, 3	failed before	4

The (incomplete) mourned sets:

$$f_1 = \varnothing, \quad f_2 = \{1\}, \quad f_3 = \{1\}, \quad f_4 = \{2, 3\}$$

Fig. 2. An example with incomplete mourned sets ($P = \{1, 2, 3, 4\}$).

THEOREM 2. *(Assuming complete mourned sets.) If $CAND_R \not\subseteq R$, where R is the set of processes with available failure information, then LAST is indeterminable.*

The proof of this theorem requires the following technical lemma. The lemma is quite general and does not depend on the completeness of mourned sets.

LEMMA 2. *For any R such that $R \subseteq P$, we have $R \cap CAND_R \neq \varnothing$.*

PROOF. This lemma is based on a simple, fundamental property of partially ordered sets, namely, a subset of a partially ordered set is also partially ordered (see Theorem A3 in the appendix). Thus any subset R of P contains a maximal element (with respect to the partial ordering FB). Let j be such a maximal element. By definition of maximal element, j failed before no other member of R; consequently, j is in CAND_R. □

PROOF OF THEOREM 2. We will show that the assumption $\text{CAND}_R \not\subseteq R$ allows us to construct two feasible extensions, fb_1 and fb_2, each implying a different LAST. In particular, we show that there exists an r such that $r \in$ LAST according to fb_1, while $r \notin$ LAST according to fb_2. Since either feasible extension could be the unknown FB relation, r's membership status in LAST is indeterminable from the available information. The choice of r is, in fact, rather straightforward: it can be any member of $R \cap \text{CAND}_R$. Lemma 2 guarantees that this set is nonempty.

Let fb_1 be equal to FB_R. Trivially, fb_1 is a feasible extension of FB_R. Moreover, if $\text{fb}_1 = \text{FB}$, then $r \in$ LAST (since according to FB_R, no process failed before r, and $\text{FB}_R = \text{fb}_1 = \text{FB}$).

To construct fb_2, first choose some j such that $j \notin R$ and $j \in \text{CAND}_R$. By the hypothesis of the theorem, such a j exists. Now let fb_2 be the feasible extension where all processes, including r, fail before j. Formally, $\text{fb}_2 = \text{FB}_R \cup \{(k, j) \mid k \in P \text{ and } k \neq j\}$. It is straightforward to verify that fb_2 is a valid feasible extension of FB_R. Since r fails before j in fb_2, $r \notin$ LAST if $\text{fb}_2 = \text{FB}$. □

The proof actually implies a stronger result than stated in the theorem: $\text{CAND}_R \subseteq R$ is necessary to determine a single member of LAST $\cap\, R$. Therefore, membership testing in LAST $\cap\, R$ is as difficult as determining the entire LAST set. In some cases, it is possible to determine a member of LAST-R without satisfying the above premise, but this is generally not useful to recovery processes.

3.1 Using Incomplete Information

With incomplete mourned sets, $\text{CAND}_{\text{LAST}}$ may contain nonmembers of LAST. This is because a member of LAST may not record all of the processes failing before it. Consequently, neither Lemma 1 nor Theorem 1 is valid in this case.

```
assume: $f_j$ for all $j \in R$ is available;
function closure(i, R) returns set-of-processes;
local tried: set-of-processes;   (* those already used in the reduction.*)
      $f_i^R$: set-of-processes;
begin
  $f_i^R$ := if ($i \in R$) then $f_i$ else $\varnothing$;
  tried := $\{i\}$;
  while ($(f_i^R - tried) \cap R \neq \varnothing$) do begin
    choose any j from $(f_i^R - tried) \cap R$;
    $f_i^R := f_i^R \cup f_j$;
    tried := tried $\cup \{j\}$;
    end;
  return ($f_i^R$);
end;
```

Fig. 3. Algorithm for calculating the *closure of* f_i with respect to set R (f_i^R).

This is illustrated in Figure 2 for $R = \{1, 4\}$, where process 4 did not detect the failure of process 1 because, for example, 4 did not attempt to communicate with 1. In this example, $\text{CAND}_R = \{1, 4\} = R$, which satisfies the premise of Theorem 1. Note however that LAST $\neq \{1, 4\}$; rather, LAST equals the singleton set $\{4\}$. The cause of the miscalculation is the incomplete mourned set of process 4; the complete mourned set includes process 1.

This suggests that the problem of determining LAST with incomplete mourned sets is harder than the problem with complete mourned sets. We will first consider a simpler problem, that of determining a single member of LAST.

Before proceeding, we need to define the *closure of a mourned set.* The closure of i's mourned set is the set of processes failing before i that can be inferred from the available data. This includes failures not recorded in f_i but implied (by transitivity) from other mourned sets. The formal definition is

Definition. The *closure of* f_i *with respect to* R, denoted f_i^R, is the set $\{j \mid j \ \text{FB}_R \ i\}$.

If $i \in R$ and mourned sets are complete, then $f_i^R = f_i$. If $i \notin R$, then by definition f_i^R is empty. The algorithm in Figure 3 efficiently computes the closure of i's mourned set without explicitly constructing FB_R.

The expression $P - f_i^R$ describes the set of processes not failing before i, given the data in R. Hence, the processes in $P - f_i^R$ failed concurrently with or after i. The next theorem tells us that the membership status of i is determinable if the failure data from this set of processes is available.

THEOREM 3. *Let i be an arbitrary process. If $i \in CAND_R$ and $(P - f_i^R) \subseteq R$, then $i \in LAST$.*

PROOF. Assume both $i \in \text{CAND}_R$ and $(P - f_i^R) \subseteq R$, but suppose for the sake of contradiction that $i \notin$ LAST. Now, $i \notin$ LAST implies that some process j received failed(i)(see remarks at the end of Section 2.2) and, as a result, $i \in f_j$. Process j can not be a member of R, for we have assumed $i \in \text{CAND}_R$, where $\text{CAND}_R = P - \bigcup_{i \in R} f_i$. Since j failed after i, j is not a member of f_i^R. Thus we have $j \in (P - f_i^R)$ and $j \notin R$. But this contradicts our assumption that $(P - f_i^R) \subseteq R$. □

Since LAST is always contained in $P - f_i^R$, the failure data of all members of LAST must be available for a membership test based on this theorem to succeed. Recall that with complete information, this is sufficient to determine all members of LAST.

A distributed procedure for testing LAST membership based on Theorem 3 is given in Figure 4. For each process i in R, an associated recovery process incrementally tests if i is a member of LAST, terminating only when membership is decided.

When failure information is complete, either the decision procedure based on Theorem 1 (Figure 1) or the procedure based on Theorem 3 (Figure 4) can be used to test membership in LAST. From a practical perspective, it is important to know how the efficacies of these two procedures compare, for if the two procedures work equally well, the one based on Theorem 3 can always be used. In this case, there would be no need to differentiate between complete and incomplete failure data.

Unfortunately, the two procedures do not work equally well. The procedure based on Theorem 3 is strictly weaker than the one based on Theorem 1: the former procedure may not be able to determine LAST even when the latter procedure can. This is because the test condition of the former ($P - f_r^R \subseteq R$) implies that of the latter ($\text{CAND}_R \subseteq R$), but not the converse. In fact, the procedure based on Theorem 3 is much weaker, as illustrated by the example in Figure 5.

Two questions remain. First, can a process detect when its mourned set is complete? If so, then processes with complete mourned sets can choose to apply the test based on Theorem 1 rather than the one based on Theorem 3. Using Theorem 1 has the additional advantage that it yields all members of LAST. Second, is there a better membership test?

We address the latter question first, since a stronger test may obviate the need to detect completeness.

THEOREM 4. *Given that $i \in CAND_R$ (and hence a candidate for LAST membership), if $(P - f_i^R) \not\subseteq R$, then the membership status of i is indeterminable.*

PROOF. The proof follows the same outline as the proof of Theorem 2. We assume that i $\in \text{CAND}_R$ and that $P - f_i^R \not\subseteq R$, and then show that it is always possible to construct two feasible extensions, one including i in LAST, the other not.

Let fb_1, the first extension, simply be FB_R. Thus, the only ordering among failures can be inferred from the available data. Trivially, fb_1 is a feasible extension, and it implies $i \in$ LAST.

The construction of fb_2, the second extension, is only slightly harder. It requires the existence of a j such that $j \in (P - f_i^R)$ and $j \notin R$. By our initial assumption, such a j exists. To construct fb_2, we add the pair (i, j) to FB_R and take the transitive closure. Thus, fb_2 is simply FB_R extended so that i fails before j. fb_2 is antisymmetric (this is easily shown), and it is transitive by construction; therefore, it is a valid feasible extension of FB_R. □

The theorem shows that deciding membership in LAST is difficult in all cases. This suggests that deciding whether f_i is complete is also difficult: an easy completeness test would imply that the membership test could be streamlined in

```
function Membership Test $(i, P, f_i)$ returns boolean;

local $j$: process;
      $R$, CAND$_R$: set-of-processes;
      $f$: array 1 . . . $P$ of set-of-processes;   (* holds mourned sets known by this process *)

begin
  $R := \{i\}$;
  CAND$_R := P - f_i$;
  broadcast message $(i, f_i)$ to $P$;
  while ($i \in$ CAND$_R \wedge$ ($P$-closure$(i, R)) \not\subseteq R$) do begin
    $(j, f[j])$ := receive( );
    $R := R \cup \{j\}$;
    CAND$_R$ := CAND$_R - f[j]$;
    end;
  if ($i \in$ CAND$_R$) then return (true)
                 else return (false)
end;
```

Fig. 4. Procedure to determine if process i is in LAST, given complete cohorts sets and incomplete mourned sets. The distributed algorithm is executed by the recovery process for i, for all $i \in R$.

Assume $f_i = f_i^R$ for all processes i and subsets R.

Let:

$f_1 = \{4, 6, 8, \cdots, n\}$; and
$f_2 = \{3, 5, 7, \cdots, n-1\}$;
$f_i = \emptyset$, for $3 \leq i \leq n$.

(Hence, process 1 failed concurrently with all odd processes and strictly after all even processes except process 2. Likewise, process 2 failed concurrently with all even processes and strictly after all odd processes except process 1.)

To determine LAST using Theorem 1 requires only that $\{1,2\} \subseteq R$. To determine LAST using Theorem 3 requires the recovery of all processes, and to determine only that $1 \in$ LAST requires the recovery of all odd processes in addition to both members of LAST.

Fig. 5. An example demonstrating that the membership test based on Theorem 3 is much weaker than the test based on Theorem 1, even with complete mourned sets.

some cases. A necessary and sufficient condition for determining completeness is given below; the proof is left as an exercise for the interested reader.

CLAIM. *To conclude that f_i is complete, it is necessary and sufficient that either $f_i^R \subseteq R$ or $\forall j (j \in f_i^R \vee i \in f_j^R)$.*

Deducing completeness from the available data is not simple and can not facilitate the determination of LAST. If R is an extensive subset of P, then $f_i = f_i^R$ strongly suggests that f_i is complete—but, in no case, except by satisfying the above condition, is this conclusive. To be useful, completeness must be an a priori premise, inferred from the architecture of process interaction.

4. A DYNAMIC ENVIRONMENT

We consider now an environment where processes may be continuously added to the process group until a total failure occurs. All of the previous results are still valid in this environment, but their application requires that each process know all members of P. Such knowledge is unreasonable for most dynamic environments; *a process can be expected to know only processes joining before its own failure.* With this constraint, the new problem confronting us is how to determine P from the available failure data.

Processes must join P in a systematic fashion in order for P, and LAST, to be determinable. Loosely speaking, the requirements for joining are: (1) P contains an operational member, and (2) the joining process makes itself "known" to all operational members. To simplify the presentation, we will use a stronger form of the second requirement: the joining process must be known to all processes failing after it joins. These requirements are captured formally in the following rules.

Inclusion rules. The event *p joins P* is allowed, if and only if

(1) $\exists q$ such that (q joins P) *happens before* (p joins P) and the failure of q does not *happen before* p joins P,
(2) $\forall q$: the failure of q does not *happen before* p joins P implies that $p \in P_q$.

Inclusion protocols—protocols satisfying these rules—are not hard to design and are in common use (see, e.g., [1]). We assume hereafter that a process joins P only by executing a proper inclusion protocol. Exempt from this requirement are the initial members of P, which may join by satisfying only Rule 2.

Rule 1 prohibits the addition of a process after a total failure has occurred. Rule 2 ensures that P_q contains all processes joining before q fails. Together these rules ensure that there exists no closed subgroup of P, that is, a subgroup of processes whose members know only other members of the subgroup. Stated formally,

LEMMA 3. *For any $S \subseteq P$, if $\bigcup_{i \in S} P_i = S$, then $S = P$.*

PROOF. (By contradiction.) Assume that the antecedent of the implication is true but the consequence is false. Let p be the member in P-S that joined before or concurrent with the other processes in P-S. By Inclusion Rule 1, we know that there exists a q joining before p joins and not failing before p joins, and by Inclusion Rule 2, we have $p \in P_q$. Since q joined before p, it must be that q is in S, and therefore, $p \in \bigcup_{i \in S} P_i$ (equivalently, $p \in S$). This contradicts our assumption. □

In the absence of closed subgroups, a test similar to the membership test of Theorem 1 can be used to determine P from the available data.

THEOREM 5. *If $(\bigcup_{i \in R} P_i - \bigcup_{i \in R} f_i) \subseteq R$, then $\bigcup_{i \in R} P_i = P$.*

PROOF. Since $f_i \subseteq P_i$, the premise can be rearranged to yield

$$\bigcup_{i \in R} P_i \subseteq \left(\bigcup_{i \in R} f_i\right) \cup R.$$

Letting S denote $(\bigcup_{i \in R} f_i) \cup R$ yields

$$\bigcup_{i \in R} P_i \subseteq S. \tag{4.1}$$

Note that if $p \in (S - R)$, then $p \in \bigcup_{i \in R} f_i$ and therefore $p \in f_i$, for some $i \in R$.

Using Inclusion Rule 2, it is straightforward to show that p failing before i implies $P_p \subseteq P_i$, and from this we conclude that

$$\bigcup_{p \in S-R} P_p \subseteq \bigcup_{i \in R} P_i.$$

Adding $\bigcup_{i \in R} P_i$ to both sides gives us

$$\bigcup_{p \in S} P_p \subseteq \bigcup_{i \in R} P_i.$$

By (4.1), we have

$$\bigcup_{p \in S} P_p \subseteq \bigcup_{i \in R} P_i \subseteq S.$$

Since $p \in P_p$, we have $S \subseteq \bigcup_{p \in S} P_p$, and therefore,

$$S = \bigcup_{p \in S} P_p = \bigcup_{i \in R} P_i.$$

By the previous lemma, this is equal to P. □

This theorem holds for incomplete as well as complete mourned sets. It immediately suggests that Theorems 1 and 3 can be extended to this environment with little modification.

COROLLARY 1. *If mourned sets are complete, then* $\bigcup_{i \in R} P_i - \bigcup_{i \in R} f_i \subseteq R$ *implies that* $\bigcup_{i \in R} P_i - \bigcup_{i \in R} f_i = LAST$.

COROLLARY 2. *If* $r \notin \bigcup_{i \in R} f_i$ *and* $(\bigcup_{i \in R} P_i - f_r^R) \subseteq R$, *then* $r \in LAST$.

PROOF. Corollary 1 follows immediately from Theorem 1 and Theorem 5. Corollary 2 follows from Theorem 3, Theorem 5, and the observation that $(\bigcup_{i \in R} P_i - f_r^R) \subseteq R$ implies $(\bigcup_{i \in R} P_i - \bigcup_{i \in R} f_i) \subseteq R$. (The observation follows from the fact that $f_i^R \subseteq \bigcup_{i \in R} f_i$.) □

Using the results in Corollaries 1 and 2, the previous decision procedures (Figures 1 and 4) can be extended to handle dynamic groups. The modified procedure for complete information is given in Figure 6.

5. DISCUSSION

In this section we discuss issues related to the maintenance of failure data, including compaction of failure data for long-lived groups. We also discuss a convenient extension: allowing processes to quit and rejoin groups.

Since the efficacy of membership testing procedures that tolerate incomplete mourned sets is much lower than those designed for complete data, it is desirable to maintain complete mourned sets and use the better procedures. Unfortunately, the maintenance of complete mourned sets is not straightforward.

Consider the following "obvious" implementation of complete mourned sets. A process appends to each message it sends its mourned set. Before processing a newly received message, a process conjoins the appended mourned set with its mourned set. This scheme has a singular deficiency: it does not work when a process fails after detecting a failure but before forwarding the observation to all of its cohorts. For example, let process 2 detect the failure of process 1. Now 2 sends a message to 3 but fails before sending one to 4. Process 4 eventually detects the failure of 2 by the standard means—a timeout message. At this point,

```
procedure $DetermineLast(i, P_i, f_i)$;
local $j$: process;
      $R, P', f'$: set-of-processes;
      $known, failed$: set-of-processes;
begin
  $R := \{i\}$;
  $known := P_i$;
  $failed := f_i$;
  broadcast message $(i, P_i, f_i)$ to $P$;
  while $(known - failed \not\subseteq R)$ do begin
    $(j, P', f') :=$ receive( );
    if $(j \notin known)$ then send $(i, P_i, f_i)$ to $j$;
    $R := R \cup \{j\}$;
    $known := known \cup P'$;
    $failed := failed - f'$;
    end;
  print $\mathrm{CAND}_R$;
end;
```

Fig. 6. The distributed algorithm executed by the recovery process for *i*. Cohorts sets are incomplete; mourned sets are complete.

4's mourned set should contain 1 and 2, but instead it contains only 1. (Note that this is the example of Figure 2.) The problem with the proposed implementation is that mourned sets cannot be appended to failure notices (in this case, the timeout). Similar problems arise in implementations based on logical clocks.

Strictly speaking, complete mourned sets are unattainable in any asynchronous system. However, they can be approximated to an acceptable degree in most systems. A brute-force approximation is to have a process send failure notices to all cohorts and wait for acknowledgements whenever it first learns of a failure. The notices must be sent and acknowledgments received prior to any changes in the task state. This ensures that mourned sets are complete with respect to the set of events changing the task state. More cost-effective approaches to maintaining complete mourned sets exist, but they generally rely on a lower level failure detection and failure data propagation facility (see [1]).

With long-lived tasks, the process group can be expected to grow to an unwieldy size. Eventually, the failure data must be either compacted or truncated. Many applications naturally lend themselves to a compact representation of the failure data. A common situation is to execute the task on a fixed group of processors with one operational member on each operational processor. Upon detecting the failure of its group member, a processor creates a new process, which then joins the group. Thus, a sequence of member processes with disjoint operational periods exists at each processor. By cleverly naming the processes of a sequence, it is possible to represent explicitly the status of the most current member (i.e., the most recently included member), and to infer the names and the statuses of the previous members. This can be accomplished, for example, by a two-part process name: the first part is a sequence identifiers, and the second part is the process's position in the sequence.

An alternative to clever naming is periodically to discard some of the failure data. An obvious strategy is to store only $P_i - f_i$ for each member i. The set difference is attractive because its size is bounded by the multiprogramming level of the task. However, vital information is lost: decision procedures as effective as the ones proposed herein do not exist for this data.

It is still possible to prune P_i and f_i without compromising Corollaries 1 and 2. Each member i need only maintain information on cohorts whose operational periods overlap with its own operational period. The formal statement of this requirement is

$$P_i \supseteq \{j \mid \neg((j\text{'s failure} \rightarrow i\text{'s inclusion}) \text{ or } (i\text{'s failure} \rightarrow j\text{'s inclusion}))\}.$$

This new requirement invalidates Lemma 3. Fortunately, Theorem 5 and Corollaries 1 and 2 are still valid, but they require new proofs since Lemma 3 is used in their current proofs. Pruning the P_i's and the f_i's does not discard useful information and, consequently, does not diminish the efficacy of the decision procedure given in Fig. 6. If LAST is determinable with full failure data from the processes in R, then LAST is determinable with truncated failure data from R.

We conclude this discussion with a simple extension for allowing a process to repeatedly join and quit a group. A process can quit a group by simulating a failure. To do so, it must notify at least one operational member, and to expedite the determination of LAST, all operational members should be notified. Moreover, it must notify all operational members if complete mourned sets are to be maintained. If a process is allowed to rejoin a group after quitting, then *member identifiers* must be substituted for process names in the foregoing discussion. Each time a process rejoins, it is assigned a new member identifier that is unique for the lifetime of the group. Unique member identifiers can easily and cheaply be generated by augmenting the process name with a count of the number of times the process has joined the group.

APPENDIX. PARTIAL ORDERINGS

In this appendix we define *partial ordering* and state several useful properties.

A *(strict) partial ordering* on a set E is an irreflexive, antisymmetric, and transitive binary relation (a set of ordered pairs) on $E \times E$. Throughout, we let R denote an arbitrary partial ordering on E. As a shorthand, we say that *R partially orders E*. To denote $(e_1, e_2) \in R$, we use the infix notation $e_1 R e_2$.

Element e, $e \in E$, is *a maximal element with respect to R* if and only if there exists no other element e', $e' \in E$, such that $e R e'$. (Whenever R is obvious from context, we will drop the phrase "with respect to R.") Two useful theorems concerning maximal elements are

THEOREM A1. *Let R be a partial ordering on E. If E is nonempty and finite, then E contains a maximal element with respect to R.*

THEOREM A2. *Let R be a partial ordering on a finite set E. For every element e in E, either e is a maximal element with respect to R or there exists a maximal element e_{max} such that $e R e_{max}$.*

Both properties can be proved from the above definitions.

Let S be a binary relation, not necessarily a partial ordering, on $E \times E$ and let E' be a subset of E. The *restriction of S to E'* is the set of ordered pairs in S both of whose elements are in E'. If S is a partial ordering, then we have the following useful result.

THEOREM A3. *If S partially orders E, then the restriction of S to E' partially orders E'.*

In addition, if e is a maximal element of E (with respect to S) and $e \in E'$, then e is a maximal element of E' with respect to the restriction of S to E'.

To simplify the notation, the term "the restriction of" is often dropped. We use the term "S partially orders E'" with the implicit understanding that "the restriction of S to E' partially orders E'" is meant.

ACKNOWLEDGMENTS

I am indebted to Nat Goodman for suggesting the descriptive terminology, in particular, "mourned sets." David Wright and the anonymous referees suggested a number of improvements in the presentation. Fred Schneider deserves special thanks for carefully reading and critiquing multiple drafts of this paper.

REFERENCES

1. GOODMAN, N., SKEEN, D., CHAN, A., DAYAL, U., FOX, S., AND RIES, D. A recovery algorithm for a distributed database system. In *Proceedings of the 2nd Symposium on the Principles of Database Systems* (Atlanta, Ga., Mar. 21-23). ACM, New York, March 1983, pp. 8-15.
2. HAMMER, M., AND SHIPMAN, D. Reliability mechanisms for SDD-1: A system for distributed databases. *Trans. Database Syst. 5*, 4 (Dec. 1980), 431-466.
3. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558-565.
4. WALTER, B. A robust and efficient protocol for checking the availability of remote sites. In *Proceedings of 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (Pacific Grove, Calif., Feb. 1982), pp. 45-68.

Received March 1983; revised August 1984; accepted November 1984.