DETERMINING THE LAST PROCESS TO FAIL

Dale Skeen

TR 82-496
February 1983

Department of Computer Science
Cornell University
Ithaca, NY 14853

Determining the Last Process to Fail

Dale Skeen

Computer Science Department
Cornell University
Ithaca, New York 14853

ABSTRACT

A *total failure* occurs whenever all processes cooperatively executing a distributed task fail before the task's completion. A frequent prerequisite for recovery from a total failure is the identification of the last group (*LAST*) of processes concurrently failing. Herein, we derive necessary and sufficient conditions for computing *LAST* from the local failure data of recovered processes. These conditions are easily translated into decision procedures for *LAST* membership using either complete or incomplete failure data. The choice of failure data itself is dictated by two requirements: (1) it can be cheaply maintained, and (2) maximum fault-tolerance is afforded in the sense that the expected number of recoveries required for identifying *LAST* is minimized.

1. Introduction

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

This problem arises in several contexts in highly fault-tolerant distributed systems, including the following two contexts from database systems: transaction management and the management of replicated data. A transaction, which is by definition an atomic action, is managed by a group of ***transaction coordinators***, where one is designated the primary and the remainder are considered backups [Hammer 80, Goodman 83]. When the primary fails, any backup can assume its responsibilities. A total failure in this context occurs when all coordinators fail. When such a failure occurs, the last coordinator alive must be identified in order to terminate the transaction safely. Similarly, copies of replicated data are maintained by a confederation of processes known as data managers. Assuming access to the data is allowed whenever any data manager is operational, recovery from a total failure requires identifying the last 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 local failure information, which is "available" whenever the processor containing that information is operational. We require that the failure data be maintainable with little overhead and yet provide a high degree of fault-tolerance in the sense that the expected number of recovered processes required for identifying $LAST$ is small. There are two subtle aspects of the problem: (1) in many situations, the failure information is necessarily incomplete, and (2) process groups are frequently dynamic – member processes are continuously added and deleted.

The 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 then shows how to infer $LAST$ when each process knows the identities of all its cohorts (i.e. other processes in the same group). Section 4 extends these results where each process knows only a subset of its fellow cohorts. Section 5 discusses implementation issues, including how to prune failure data for long-lived groups.

2. Background

2.1. The Environment

A process initially occupies the *operational* state and eventually makes a single transition to the *failed* state. Note that a process fails only once. Process failures are benign: a process fails by abruptly halting, and while it is failed, it does nothing. This is a standard assumption for our target applications, including database systems, and is justified by: (1) almost all failures can be modeled as above,[1] and (2) software tolerating less well-behaved failures is prohibitively expensive.

A failure is detectable by any process attempting to communicate with the failed process. For concreteness, we use the standard mechanism for detecting failures, a ***timeout***, which can be thought of as a special message sent only by failed processes. The "recipient" of a ***timeout*** can safely assume that the "sending" process has failed.[2]

A ***process group*** is a finite, nonempty set of communicating and cooperating processes. In general, a process group is not a fixed set of processes; rather, new processes can "join" the group by executing a suitable protocol. Among other thing, the protocol makes the new member known to the other members of the group. While 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***. For convenience, we assume that a member can never "quit" a group (i.e. become a nonmember). The evolution of each process with respect to a process group is thus:

nonmember → operational member → failed member

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

Members communicate only through messages. This crucial assumption is used in the definition of event ordering (see below). To assume otherwise exacerbates the problem, complicating the definition of such basic notions as "failed before." Later, we will relax this restriction so that only operational members need communicate through messages. Since operational members normally reside at different processors and necessarily communicate only through messages, this restriction is inconsequential in practice.

---

[1] This assumes that no malicious user is directing the process.

[2] This assumption can be relaxed to where the recipient runs a protocol verifying the status of the other process whenever a failure is suspected. Status verification protocols with an arbitrarily small error tolerance can be designed.

2.2. Event Ordering

In contrast to a centralized system, event ordering in a distributed system is only partial. Accordingly, understanding a distributed system requires an elementary understanding 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* insofar as causality is preserved. Event a can affect event b only if a "occurs before" b. Within a distributed system, for an event at one process to affect an event at another process, a message must propagate from the first process to the second (possibly through intermediate processes). This message may be an explicit message or an implicit one, such as a *timeout.*

For a message-based system, Lamport defined such an ordering in [Lamport 78]. It is, of course, partial.

> **Definition** (from [Lamport 78]). The *occurs before* relation, denoted by "$\rightarrow$", on a set of events is the smallest relation satisfying the following three conditions: (1) if a and b are events at the *same* process, and a occurs before b according to some local clock, then $a \rightarrow b$, (2) if event a sends a message that is received by event b, then $a \rightarrow b$, (3) if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. Distinct events a and b are said to be *concurrent* if neither $a \rightarrow b$ nor $b \rightarrow a$.

If $a \rightarrow b$, then b could not have affected a. Concurrent events therefore can not causally affect one another.

Significant events include the joining of a process to a process group and, most importantly, failures. Although our language is at times informal, for example "i joins before j fails," all references to event ordering are based on the formal definition of "occurs before."

The *failed before* relation, denoted FB, is derived from the subrelation of "occurs before" concerned only with failures. It is defined over processes rather than events.

> **Definition.** Let i and j be two processes. $i \ FB \ j$ if and only if i's failure $\rightarrow$ j's failure.

Two processes, i and j, concurrently fail if neither $i \ FB \ j$ nor $j \ FB \ i$. That FB is a partial ordering clearly follows from the fact that "occurs before" is a partial ordering.

We can now formally define $LAST$. $LAST$ is defined only with respect to a process group P that has experienced a total failure.

> **Definition.** $LAST = \{ j \mid \neg \exists i : i \in P : (j \ FB \ i) \}$.[3]

LAST is the set of maximal elements of P with respect to the partial ordering *FB*.

2.3. Failure Information

Each process in a group maintains failure information about its cohorts, which is readable by subsequent recovery processes. However, at any given time, only the information from a subset of the processes in a given group is likely to be accessible – the remaining being inaccessible because, for example, the physical processors holding the information are inoperative.

Each process i in group P maintains on nonvolatile storage two sets:

(1) P_i – 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. We say that i "mourns" the members of f_i.

P_i may not equal P because, for example, processes can join P after i's failure. 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.

If f_i contains all processes failing before i, i.e., $f_i = \{j \mid j \; FB \; i\}$, then f_i is said to be ***complete***; otherwise, it is ***incomplete.*** If all mourned sets are complete, then the failure data is said to be complete.

Failure data is assumed to be correct but not necessarily complete. Minimally, f_i contains all failures directly detected (through a ***timeout***) by i. It does not contain i; a process can not record its own failure. Minimally, P_i contains i and f_i. These constraints are necessary to ensure that *LAST* is reconstructible from the failure data; specifically, $\bigcup_{i \in P} P_i - \bigcup_{i \in P} f_i = LAST$.

It is from the collection of available P_i's and f_i's that *LAST* is determined. An initial, nonempty collection determines a set of candidates for *LAST*; as more data becomes available, this set shrinks. Our primary task is to determine when sufficient data is available to conclude that the candidate set equals *LAST*. A weaker result, which is satisfactory for many applications, is conclusively identifying a single member of *LAST*.

Not all information available to a member process is recorded; namely, the order of observed failures is not recorded. The question naturally arises as to whether ordering the elements within mourned sets can facilitate the determination of *LAST*. Intuitively, this seems unlikely since we are interested only in identifying maximal elements of "fails before." This is indeed the case as is formally argued in a later section.

The available failure data induces a partial ordering among process failures. Even when all data is available, this ordering is generally weaker than the failed-before relation because of the lack of ordering within and the possible incompleteness of mourned sets. Nonetheless, this

---

[3]Notation: in a quantified expression, the term between colons (in this case, $i \in P$) defines the range of the quantified variable.

induced ordering will be instrumental in proving algorithms for calculating $LAST$. We denote it 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 $i\ FB_R\ j$, and

(2) if $i\ FB_R\ j$ and $j\ FB_R\ k$, then $i\ FB_R\ k$.

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.

Clearly $FB_R \subseteq FB$. If the failure data is complete, then $FB_P = FB$; but this is generally not true for incomplete data. It is a simple exercise to show that the maximal elements of FB_P are also the maximal elements of FB, which are the members of $LAST$ by definition. Thus, $LAST$ is always determinable even if FB is not.

We can view FB_R as defining a set of possible candidates for the unknown FB relation. These candidates, called *feasible extension of* FB_R, are 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,

(2) fb extends FB_R (i.e. $FB_R \subseteq fb$).

In addition, if FB_R is complete than fb must also satisfy:

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

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

3. Determining the Last Group to Fail

We assume in this section that P is fixed *a priori* and known to every member process – a restriction that is relaxed in the next section. As before, R denotes the subset of processes with available failure data.

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

3.1. Using Complete Information

A useful result is:

Lemma 1. With complete mourned sets, $P - \bigcup_{i \in LAST} f_i = LAST$

Proof. We first assert that either a process i is in $LAST$ or it failed before some member of $LAST$. The argument is straightforward. If $i \notin LAST$, then $\exists\, i_1$ such that i FB i_1. Now either $i_1 \in LAST$ or $\exists\, i_2$ such that i_1 FB i_2, and so on. Since P is finite, this chain must terminate. The last process, say i_k, must be in $LAST$. By transitivity, i FB i_k and therefore $i \in f_{i_k}$ (recall that mourned sets are assumed to be complete). This proves that if a process is not in $LAST$, it must be in $\bigcup_{i \in LAST} f_i$. On the other hand, a member of $LAST$ is clearly not in any mourned set and consequently it must be $P - \bigcup_{i \in LAST} f_i$. □

With complete data, $LAST$ is determinable if failure data from all its members is available. This is not surprising, but it is useful only if there is an effective to test whether the data for all of $LAST$'s members is available. The next theorem suggests a simple test.

Theorem 1. Let R be an arbitrary subset of P and assume the completeness of mourned sets. If $P - \bigcup_{i \in R} f_i \subseteq R$, then $P - \bigcup_{i \in R} f_i = LAST$.

Proof. From basic definitions, it clearly follows that $LAST \subseteq P - \bigcup_{i \in R} f_i$. By assumption, $P - \bigcup_{i \in R} f_i \subseteq R$; therefore, $LAST \subseteq R$. This together with Lemma 1 implies $P - \bigcup_{i \in R} f_i = P - \bigcup_{i \in LAST} f_i = LAST$. □

We now argue that the condition in the above theorem is "necessary" – necessary not in the sense that the conclusion ($P - \bigcup_{i \in R} f_i = LAST$) implies the premise ($P - \bigcup_{i \in R} f_i \subseteq R$), which it obviously does not, but rather in the sense that the premise is the weakest possible. If the premise is not satisfied, then LAST membership, at least for one process, is indeterminate.

Theorem 2. (Assuming complete mourned sets.) $P - \bigcup_{i \in R} f_i \subseteq R$, where R is the set of processes with "available" failure information, is "necessary" in order to determine $LAST$.

The proof requires the following lemma. The lemma itself is quite general and does not depend on the completeness of failure data.

Lemma 2. For any R, such that R is a subset of P, we have $(P - \bigcup_{i \in R} f_i) \cap R \neq \phi$.

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. Thus any subset R of P contains a maximal element (with respect to the ordering FB). Let j be such a maximal element. By definition, j failed before no other member of R; consequently, j is in $P - \bigcup_{i \in R} f_i$. □

Proof of Theorem 2. (By contradiction.) Assume that $P - \bigcup_{i \in R} f_i$ is not contained in R. We show that it is always possible to construct at least two feasible extensions, each implying a different $LAST$.

Let r be a member of $(P - \bigcup_{i \in R} f_i) \cap R$. By Lemma 2, such a r must exist. Let fb_1, the first feasible extension, be equal to FB_R. Trivially, fb_1 is a feasible extension of FB_R. Moreover, if $fb_1 = FB$, then $r \in LAST$. Now let fb_2, the second extension, be equal to $FB_R \cup \{(k,j) \mid k \in P \text{ and } k \neq j\}$ for some $j \in (P - \bigcup_{i \in R} f_i) - R$. By assumption, j exists. It is straightforward to verify that fb_2 is also a feasible extension of FB_R. In fb_2 all processes, including r, fail before j; hence, if $fb_2 = FB$, then $r \notin LAST$. □

The proof actually implies a stronger result: $P - \bigcup_{i \in R} f_i \subseteq R$ is "necessary" in order to determine a single member of $LAST \cap R$. Therefore, membership testing in $LAST \cap R$ is difficult as determining the entire $LAST$ set. In rare 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.

Notice that the argument in the above proof is valid even if each process maintains the partial ordering of all failures occurring before its own. Partially-ordered mourned sets can not facilitate determination of $LAST$.

3.2. Using Incomplete Information

With incomplete mourned sets, $P - \bigcup_{i \in LAST} f_i$ subsumes, rather than equals, $LAST$. Consequently, neither Lemma 1 nor Theorem 1 apply. This is illustrated in figure 1 when $R = \{1,4\}$. In this case $P - \bigcup_{i \in R} f_i = \{1,4\} = R$, which satisfies the premise of Theorem 1. Note however that $LAST \neq \{1,4\}$; instead, $LAST$ equals the singleton set $\{4\}$. The cause of the miscalculation is the incomplete mourned set of process 4; the complete set includes process 1.

The above remarks coupled with the example strongly suggest that determining $LAST$ with incomplete mourned sets is a fundamentally harder problem. Since it is no longer obvious that determining a single member of $LAST$ is as difficult as determining all members, we consider the former problem first.

The observed ordering of failures:

1	*failed before*	2
2	*failed concurrent with*	3
2, 3	*failed before*	4

The (incomplete) mourned sets:

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

Figure 1. An example with incomplete failure data ($P=\{1,2,3,4\}$).

Before proceeding, it is important to define the ***closure of a mourned set.*** The closure for an arbitrary i includes all process failures preceding i's failure that can be inferred from the available data. This includes failures not recorded in f_i but implied through 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 $\{j \mid j \ FB_R \ i\}$.

If mourned sets are complete, then $f_i^R=f_i$. If $i \notin R$, then f_i^R is empty. The closure can be efficiently computed without explicitly constructing FB_R by the algorithm in figure 2.

The closure of i's mourned set is the maximum failure information available to i and is used in the following simple membership condition.

Theorem 3. If $r \notin \bigcup_{i \in R} f_i$ and $(P-f_r^R) \subseteq R$, then $r \in LAST$.

Proof. (By contradiction.) Assume both $r \notin \bigcup_{i \in R} f_i$ and $(P-f_r^R) \subseteq R$, but suppose that $r \notin LAST$. Consequently, there exists a j directly observing the failure of r. Of course, $r \in f_j$. Since we assume $r \notin \bigcup_{i \in R} f_i$, j can not be a member of R. Since j failed after r, j is not a member of f_r^R. In other words, $j \in (P-f_r^R)$ but $j \notin R$. But this contradicts our assumption that $(P-f_r^R) \subseteq R$. □

Comment. Compute f_i^R given i, R, and f_j for all $j \in R$.

Declarations.

TRIED - set of processes already used in the reduction.

Algorithm.

$f_i^R :=$ if $(i \in R)$ then f_i else $\emptyset$;

TRIED $:= \{i\}$;

while $(f_i^R - \text{TRIED}) \cap R \neq \phi$ do

choose any j from $(f_i^R - \text{TRIED}) \cap R$;

$f_i^R := f_i^R \cup f_j$;

TRIED $:=$ TRIED $\cup \{j\}$;

end;

Figure 2. Algorithm for calculating the *closure of* f_i with respect to set R.

Since $LAST$ is always contained in $P - f_r^R$, all members of $LAST$ must recover in order for the membership test to succeed. Recall that with complete information, this is sufficient to determine all members of $LAST$.

Note that $P - f_r^R \subseteq R$ implies $P - \bigcup_{i \in R} f_i \subseteq R$. This is obvious from the definition of f_r^R and it means that the premise of Theorem 3 implies the premise of Theorem 1. The converse, though, is false; therefore, Theorem 3 is strictly weaker than Theorem 1. In fact, Theorem 3 is much weaker since its premise may fail arbitrarily long after the premise in Theorem 1 is satisfied – even with complete failure data. This is illustrated in figure 3.

Two questions remain. First, can a process detect when its alive set is complete? If so, then processes with complete alive sets can choose to apply the test in Theorem 1 rather than the one in Theorem 3. Using Theorem 1 has the additional advantage that it yields all

Assume:

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

Let:

$f_1 = \{4, 6, 8, \cdots, n\}$; and

$f_2 = \{3, 5, 7, \cdots, n-1\}$;

$f_i = \emptyset$, where $i \neq 1$ or 2.

(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*.

Figure 3. An example demonstrating that the membership test in Theorem 3 is much weaker than the test in Theorem 1. The failure data is complete.

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 $r \notin \bigcup_{i \in R} f_i$ (and hence a candidate for membership in *LAST*), $(P - f_r^R) \subseteq R$ is "necessary" for deciding set membership.

Again, "necessary" is used in the same sense as in Theorem 2. The proof follows the same basic outline as the proof of that theorem.

Proof. (By contradiction.) Assume that $r \notin \bigcup_{i \in R} f_i$ and that $P - f_r^R \not\subseteq R$. We show that it is always possible to construct two feasible extensions, one including r 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 clearly implies $r \in LAST$.

The construction of fb_2, the second extension, is only slightly harder. It requires the existence of a j such that $j \in (P - f_r^R)$ and $j \notin R$. By assumption, j exists. Let fb_2 be the transitive closure of FB_R augmented with the pair (r,j). In this ordering r *fails before* j. fb_2 is easily shown to be antisymmetric, and it is transitive by construction; therefore, fb_2 is a correct feasible extension of FB_R. $\square$

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 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 $\forall j : j \notin f_i^R : (i \in f_j^R)$.

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.

3.3. Towards an Implementation

Consider the following "obvious" implementation of mourned sets. Whenever a process receives a timeout from another process, say j, it immediately adds j to its mourned set and then appends the notice "j has failed" to the next message to each process. The receiver of such a piggybacked notice must first add j to its mourned set before it can safely act on the message. The purpose of piggybacking the failure notice, rather than sending it directly, is to ensure that routing delays do not postpone its arrival until after the arrival of a subsequent message.

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. A simple example illustrates. Process 1 fails causing process 2 to fail. Afterwards, process 3 detects the failure of 2 but not of 1 (perhaps 3 rarely communicates with 1), and then itself fails. Clearly, process 3's mourned set should not contain process 1, but it does. Although such a simple scenario is unlikely to present problems, more complex failure sequences can. An interesting example involving commit protocols is given in [Skeen 82].

The fundamental problem with this and any other implementation of mourned sets is that failure notices can not be piggybacked on top of timeout messages. Cascaded failures can not propagate failure information, necessarily leaving mourned sets incomplete. Similar problems arise in timestamp approaches unless timeouts can be assigned consistent timestamps. In most environments this is either difficult or expensive.[4]

The above does not imply that the results for complete mourned sets are useless. Systems can be designed that both propagate failure information in a timely fashion and prohibit critical state transitions during its propagation (see [Goodman 83]). In such systems, mourned sets can be considered complete.

4. A Dynamic Environment

Consider now an environment where processes are continuously spawned and added to the process group until a total failure occurs. All of the previous results hold in this environment, but their application requires knowledge of P (i.e., $P_i = P$). Such knowledge is unreasonable and generally impossible in a dynamic environment; a process can be expected to know only processes joining before its own failure. Furthermore, checking $P_i = P$ is problematic.

Processes must join P in a systematic fashion in order for P, and hence $LAST$, 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. 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 *occurs before* p joins P and the failure of q does not *occur before* p joins P,

(2) $\forall q$: p joins P not *occurring before* the failure of q implies that $p \in P_q$.

Inclusion protocols – protocols satisfying these rules – are not hard to design and are in common use (see, for example, [Goodman 83]). 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. This is a sound operational requirement for most distributed tasks. Rule 2 ensures that P_p contains all processes joining before p fails. This is stronger than necessary – P_p need only contain members whose operational period overlapped with p's operational period – but it's present form simplifies the presentation. Together these rules ensure that there are no closed subgroups of P – a subgroup of processes whose members know only other members of the subgroup.

---

[4]This requires that either the rate of divergence of local clocks be bounded and known (which is expensive) or that the system be able to determine the timestamp of the last message sent by the failed process (which is difficult).

Lemma 3. For any $X \subseteq P$, $\bigcup_{x \in X} P_x = X$ implies $X = P$.

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

With this property, 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 $\bigcup_{i \in R} P_i \subseteq (\bigcup_{i \in R} f_i) \cup R$. Let X denote $(\bigcup_{i \in R} f_i) \cup R$. Note that if $x \in X-R$, then $x \in \bigcup_{i \in R} f_i$ and therefore x failed before some i, $i \in R$. Using Rule 2, it is easily shown that *x failing before i* implies $P_x \subseteq P_i$. Therefore, $\bigcup_{x \in X-R} P_x \subseteq \bigcup_{i \in R} P_i$. Conjoining $\bigcup_{i \in R} P_i$ to both sides yields $\bigcup_{x \in X} P_x \subseteq \bigcup_{i \in R} P_i$. We now have that $\bigcup_{x \in X} P_x \subseteq \bigcup_{i \in R} P_i \subseteq X$, and it is obvious that $X \subseteq \bigcup_{x \in X} P_x$; therefore, $X = \bigcup_{x \in X} P_x = \bigcup_{i \in R} P_i$. Using the previous lemma, we can now assert $X = \bigcup_{r \in R} P_r = P$. □

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 failure information is 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 $r \in LAST$.

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 $\bigcup_{i \in R} P_i - \bigcup_{i \in R} f_i \subseteq R$.

5. Discussion

Theorems 1 and 3 and Corollaries 1 and 2 yield easily implemented decision procedures for *LAST* membership in the four environments discussed. The procedures can be implemented in either a centralized or decentralized fashion. [Goodman 83] discusses a decentralized implementation assuming complete information in a dynamic environment.

The operational overhead for recoverability from total failure consists of per process overhead of maintaining failure data about all group members and of the systemwide overhead of propagating failure data. Both types of overhead are continuously incurred over the lifetime of the process group, whether or not a total failure occurs. Failure data can be cheaply maintained in nonvolatile storage for a reasonable number of members; however, data compaction becomes a problem when the number of failed members becomes large. The dominate cost normally is that of failure detection and failure data propagation.

With long-lived tasks, the process group can be expected to grow to an unwieldy size. The failure data must then 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 ids and the statuses of the previous members. This can be accomplished, for example, by a two part process id: the first part is a sequence id, the second part is process's position in the sequence.

An alternative to clever naming is to periodically 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 Theorem 4; consequently, Corollaries 1 and 2 require new and more intricate 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 procedures. If $LAST$ is determinable with full failure data from the processes in R, then $LAST$ is determinable with truncated failure data from R.

The cost of failure detection and failure data propagation depends on the completeness of the information. An obvious tradeoff exists: it is costlier to maintain more complete failure data, but it expedites determination of $LAST$. Minimally, a process need only know about failures of the processes with whom it is in communication. This coincides with the minimum requirement for any fault-tolerant system, irrespective of how it handles total failures. More complete information can be achieved by piggybacking failure notices on regular messages. The additional cost is very much a function of system design, specifically, the nature of the message interface. Complete failure information, strictly speaking, is not achievable; however, systems can be designed that prohibit critical task processing during the propagation of failure notices to all operational members. Such systems effectively maintain complete data. The cost of synchronizing the propagation of failure data with task execution is substantial and probably not justifiable for total failure recovery alone. On the other hand, such synchronization may be desirable for other reasons.

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 cohort, and it is probably good system design to notify all operational cohorts. It must notify all if complete failure data is to be maintained. If a process is allowed to rejoin a group after quitting, then *member ids* must be substituted for process ids in the foregoing discussion. Each time a process rejoins, it is assigned a new member id that is unique for the lifetime of the group. Unique member ids can easily and cheaply be generated by augmenting the process id with a count of the number of times the process has joined.

References

[Goodman 83] Nathan Goodman et al., "A Recovery Algorithm for a Distributed Database System," *Proceedings of the Second Symposium on the Principles of Database Systems*, March 1983.

[Hammer 80] Michael Hammer and David Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *Transactions on Database Systems*, 5, 4, December 1980, pp. 431-466.

[Lamport 78] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," pp. 558-565, *Communications of the ACM*, 21, 7, July 1978.

[Skeen 82] Dale Skeen, "Crash Recovery in a Distributed Database System," Ph.D. Thesis, University of California, Berkeley, May 1982 (ERL Memo M82/45).

APPENDIX

Partial Orderings.

A partial ordering Φ over a set E is an irreflexive, antisymmetric, and transitive binary relation on E. The notation $e_1 \Phi e_2$ is a standard synonym for $(e_1, e_2) \in \Phi$.

Element $e \in E$ is ***maximal*** (with respect to Φ) if and only if there exists no other element $e' \in E$ such that $e \Phi e'$. If E is finite and nonempty, then E contains at least one maximal element.

Let E' be a subset of E. Any partial ordering over E is also a partial ordering over E'. Moreover, if e is a maximal element of E and $e \in E'$, then e is also a maximal element of E'.