Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing

David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas

Abstract

In a distributed system using message logging and checkpointing to provide fault tolerance, there is always a unique maximum recoverable system state, regardless of the message logging protocol used. The proof of this relies on the observation that the set of system states that have occurred during any single execution of a system forms a lattice, with the sets of consistent and recoverable system states as sublattices. The maximum recoverable system state never decreases, and if all messages are eventually logged, the domino effect cannot occur. This paper presents a general model for reasoning about recovery in such a system and, based on this model, an efficient algorithm for determining the maximum recoverable system state at any time. This work unifies existing approaches to fault tolerance based on message logging and checkpointing, and improves on existing methods for optimistic recovery in distributed systems.

1 Introduction

Message logging and checkpointing can be used to provide an effective fault-tolerance mechanism in a distributed system in which all process communication is through messages. Each message received by a process is logged on stable storage [5], and each process is occasionally checkpointed to stable storage, but no coordination is required between the checkpoints of different processes. Between received messages, the execution of each process is assumed to be deterministic.

The protocols used for message logging are typically *pessimistic*. With these protocols, each message is synchronously logged as it is received, either by blocking the receiver until the message is logged [1, 6], or by blocking the receiver if it attempts to send a new message before this received message is logged [3]. Recovery based on pessimistic message logging is straightforward. A failed process is restarted from its last checkpoint, and all messages originally received by this process since the checkpoint are replayed to it from the log in the same order as they were received before the failure. The process reexecutes based on these messages to its state at the time of the failure. Messages sent by the process during recovery are ignored since they are duplicates of those sent before the failure.

On the other hand, *optimistic* protocols perform the message logging asynchronously [9]. The receiver continues to execute normally, and received messages are logged later, for example by grouping several messages and writing them to stable storage in a single operation. The receiver of a message *depends on* the state of the sender, though. If the sender fails and cannot be recovered (for example, because some message has not been logged), the receiver becomes an *orphan* process, and its state must be rolled back during recovery to a point before this dependency was created. If rolling back this process causes other processes to become orphans, they too must be rolled back during recovery. The *domino effect* [7, 8] is an uncontrolled propagation of such rollbacks and must be avoided to guarantee progress in spite of failures. Recovery based on optimistic message logging must

This work was supported in part by the National Science Foundation under grant DCR-8511436 and by the Office of Naval Research under grant ONR N00014-88-K-0140.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and / or specific permission.

© 1988 ACM 0-89791-277-2/88/0007/0171 $1.50

find the "most recent" combination of process states that can be recreated, such that none of the process states is an orphan.

Optimistic message logging protocols appear to be desirable in systems in which failures are rare and failure-free performance is of primary concern. Since optimistic protocols avoid synchronization delays during message logging, performance in the absence of failures is improved. Although the required recovery procedure is then more complicated, this procedure is only required when a failure occurs.

Section 2 of this paper presents a general model for reasoning about these recovery methods in distributed systems. With this model, we show that there is always a unique maximum recoverable system state, which never decreases, and that if all messages received are eventually logged, the domino effect cannot occur. Based on this model, Section 3 describes and proves the correctness of our algorithm for determining the maximum recoverable system state at any time. The algorithm requires no additional messages in the system, and supports recovery from any number of concurrent failures, including a total failure. Our model and algorithm make no assumption of the message logging protocol used; they support both pessimistic and optimistic logging protocols, although pessimistic protocols do not require their full generality. Section 4 then relates this work to existing fault-tolerance methods published in the literature and discusses the effect of different message logging protocols on our model and algorithm. Finally, Section 5 summarizes the contributions of this work and draws some conclusions.

2 The Model

2.1 Process States

Each time a process receives an input message, it begins a new *state interval*, a deterministic sequence of execution based only on the state of the process at the time that the message is received and on the contents of the message itself. Within each process, each state interval is identified by a unique sequential *state interval index*, which is simply a count of the number of input messages that the process has received.

All dependencies of a process i on some process j can be encoded simply as the maximum index of any state interval of process j on which process i depends. This encoding is possible since the execution of a process within each state interval is deterministic and since any state interval in a process naturally also depends on all previous intervals of the same process.

All dependencies of any process i can, therefore, be represented by a *dependency vector*

$$\mathbf{d}_i = \langle \delta_* \rangle = \langle \delta_1, \delta_2, \delta_3, \ldots, \delta_n \rangle \ ,$$

where n is the total number of processes in the system. Component j of process i's dependency vector, δ_j, gives the maximum index of any state interval of process j on which process i currently depends. Component i of process i's own dependency vector is always set to the index of process i's current state interval. If process i has no dependency on any state interval of some process j, then δ_j is set to $\perp$, which is less than all possible state interval indices.

Processes cooperate to maintain their current dependency vectors by tagging all messages sent with their current state interval index and by remembering in each process the maximum index in any message received from each other process. During any single execution of the system, the dependency vector for any process is uniquely determined by the state interval index of the process. No component of the dependency vector of any process can decrease through normal execution of the process.

2.2 System States

The state of the system is the composition of the states of all component processes of the system and may be represented by an $n \times n$ *dependency matrix*. Taking the dependency vector, $\mathbf{d}_i$, of each process i in the system, the dependency matrix

$$\mathbf{D} = [\delta_{*\,*}] = \begin{bmatrix} \delta_{1\,1} & \delta_{1\,2} & \delta_{1\,3} & \ldots & \delta_{1\,n} \\ \delta_{2\,1} & \delta_{2\,2} & \delta_{2\,3} & \ldots & \delta_{2\,n} \\ \delta_{3\,1} & \delta_{3\,2} & \delta_{3\,3} & \ldots & \delta_{3\,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_{n\,1} & \delta_{n\,2} & \delta_{n\,3} & \ldots & \delta_{n\,n} \end{bmatrix}$$

can be formed, where row i, $\delta_{i\,j}$, $1 \leq j \leq n$, is the dependency vector for process i. Since component i of process i's dependency vector is always the index of process i's current state interval, the diagonal of the dependency matrix, $\delta_{i\,i}$, $1 \leq i \leq n$, shows the current state interval index of each process in the system.

Let $\mathcal{S}$ be the set of all system states that have occurred during any *single* execution of some system. The set $\mathcal{S}$ forms a partial order called the *system history relation*, in which one system state precedes another if and only if it *must* have occurred first during the execution. This relation can be expressed in terms of the state interval index of each process as shown in the dependency matrix.

Definition 1 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then

$$\mathbf{A} \preceq \mathbf{B} \iff \forall\, i\, [\alpha_{ii} \leq \beta_{ii}] \, .$$

This partial order differs from that defined by Lamport's *happened before* relation [4] in that it orders the system states that result from events rather than the events themselves, and that only state intervals (started by the receipt of a message) constitute events.

For example, Figure 1 shows a system of four communicating processes. The horizontal lines represent the execution of each process, each arrow represents a message from one process to another, and the numbers give the index of the state interval started by the receipt of each message. Consider the two possible system states $\mathbf{A}$ and $\mathbf{B}$, where in state $\mathbf{A}$, message a has been received but message b has not, and in state $\mathbf{B}$, message b has been received but message a has not. These states can be expressed by the dependency matrices

$$\mathbf{A} = \begin{bmatrix} \textcircled{2} & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & \bot & \textcircled{0} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \textcircled{1} & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & 2 & \textcircled{1} \end{bmatrix} .$$

States $\mathbf{A}$ and $\mathbf{B}$ are incomparable under the system history relation, which can be seen by comparing the circled values on the diagonals of these two dependency matrices.

2.3 The System History Lattice

A system state describes the set of messages that have been received by each process. Two system states in $\mathcal{S}$ can be combined to form their ***union*** such that each process has received all of the messages that it has in either of the two original system states. This can be expressed in terms of the dependency matrices describing these system states by choosing for each process the row that has the largest state interval index of the corresponding rows in the original matrices.

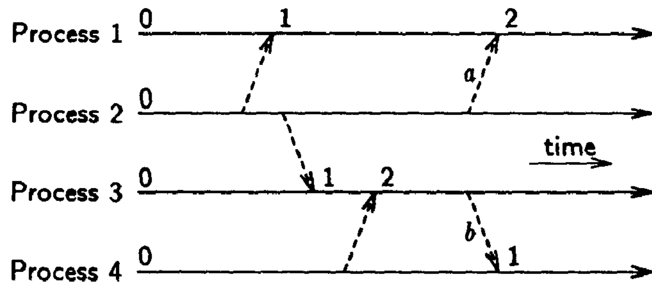

Figure 1 The system history partial order

Definition 2 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then the ***union*** of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \cup \mathbf{B} = [\gamma_{**}]$,

$$\forall\, i \left[\gamma_{i*} = \begin{cases} \alpha_{i*} & \text{if } \alpha_{ii} \geq \beta_{ii} \\ \beta_{i*} & \text{otherwise} \end{cases} \right] .$$

Likewise, the ***intersection*** of two system states in $\mathcal{S}$ can be formed such that each process has received only those messages that it has in both of the two original system states. This can be formed from the dependency matrices describing these states by choosing for each process the row that has the smallest state interval index of the corresponding rows in the original matrices.

Definition 3 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then the ***intersection*** of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \cap \mathbf{B} = [\delta_{**}]$,

$$\forall\, i \left[\delta_{i*} = \begin{cases} \alpha_{i*} & \text{if } \alpha_{ii} \leq \beta_{ii} \\ \beta_{i*} & \text{otherwise} \end{cases} \right] .$$

Continuing the example of Section 2.2 in Figure 1, the union and intersection of states $\mathbf{A}$ and $\mathbf{B}$ can be formed by choosing the proper rows from these two matrices to get

$$\mathbf{A} \cup \mathbf{B} = \begin{bmatrix} 2 & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & 2 & 1 \end{bmatrix} \quad \mathbf{A} \cap \mathbf{B} = \begin{bmatrix} 1 & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & \bot & 0 \end{bmatrix} .$$

The following theorem introduces the ***system history lattice*** formed by the set of system states that have occurred during any ***single*** execution of some system, ordered by the system history relation.

Theorem 1 The set $\mathcal{S}$, ordered by the system history relation, forms a lattice. For any $\mathbf{A}, \mathbf{B} \in \mathcal{S}$, the ***least upper bound*** of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \cup \mathbf{B}$, and the ***greatest lower bound*** of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \cap \mathbf{B}$.

Proof Straightforward from the construction of system state union and intersection in Definitions 2 and 3. □

2.4 Consistent System States

A system state is *consistent* if and only if all messages received by all processes have either already been

sent in the state of the sending process or can deterministically be sent by that process in the future. Since process execution within a state interval is deterministic, any message sent before the end of the current state interval can deterministically be sent, but messages sent after this cannot be. Only a consistent system state would be *possible* to be reached through normal execution of the system from its initial state, if an instantaneous snapshot of the entire system could be observed [2].

Any messages shown by the system state to be sent but not yet received do not cause the system state to be inconsistent. These messages can be handled by the normal mechanism for reliable message delivery, if any, used by the underlying system. In particular, suppose such a message m was received by some process i after the state of process i was observed to form the system state, and suppose process i then sent some message n (such as an acknowledgement of message m), which could show this receipt. If message n has been received in this system state, the state will be inconsistent because message n (not message m) is shown as having been received but not yet sent. If message n has not been received yet, no effect of either message can be seen in the system state, and it is thus still consistent.

If a system state is consistent, then no process depends on a state interval of the sender greater than the sender's current state interval in the dependency matrix. For each column j of the dependency matrix, no element in that column may be larger than the element on the diagonal of the matrix.

Definition 4 If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathcal{S}$, $\mathbf{D}$ is *consistent* if and only if

$$\forall\, i,j \; [\delta_{ij} \leq \delta_{jj}] \; .$$

Let the set $\mathcal{C} \subseteq \mathcal{S}$ be the set of *consistent* system states that have occurred during any single execution of some system. Thus,

$$\mathcal{C} = \{\, \mathbf{D} \in \mathcal{S} \mid \mathbf{D} \text{ is consistent} \,\} \; .$$

For example, consider the system of three processes whose execution is shown in Figure 2. The state of each process here is observed where the curve crosses the execution line for that process, and the resulting system state is represented by the dependency matrix

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} 1 & \textcircled{4} & \perp \\ 0 & \textcircled{2} & 0 \\ \perp & 2 & 1 \end{bmatrix} \; .$$

This system state is not consistent since process 1 has received a message (to begin state interval 1) from process 2 that has not been sent yet by process 2 and cannot be deterministically sent in the future. This inconsistency is shown in the dependency matrix since δ_{12} is greater than δ_{22}.

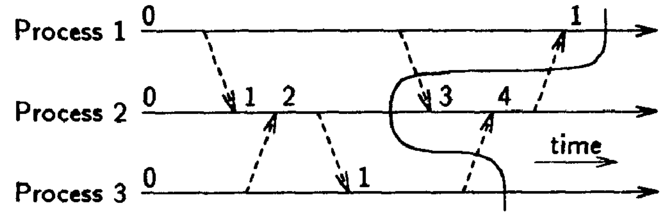

Figure 2 An inconsistent system state

Lemma 1 The set $\mathcal{C}$ forms a sublattice of the system history lattice.

Proof It suffices to show that for any $\mathbf{A}, \mathbf{B} \in \mathcal{C}$ $\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$ and $\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$. Let $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$.

($\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$): Let $\mathbf{C} = [\gamma_{**}] = \mathbf{A} \cup \mathbf{B}$. In each column j of $\mathbf{C}$, either $\gamma_{ij} \leq \alpha_{jj}$ or $\gamma_{ij} \leq \beta_{jj}$ for all i, since $\mathbf{A} \in \mathcal{C}$ and $\mathbf{B} \in \mathcal{C}$. Since $\gamma_{jj} = \max(\alpha_{jj}, \beta_{jj})$, $\gamma_{ij} \leq \gamma_{jj}$ for all i as well. Therefore, $\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$.

($\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$): Let $\mathbf{D} = [\delta_{**}] = \mathbf{A} \cap \mathbf{B}$. By Definition 3 and since no element in the dependency vector for any process ever *decreases* as the process executes $\delta_{ij} = \min(\alpha_{ij}, \beta_{ij})$, for all i and j. This implies that $\delta_{ij} \leq \alpha_{ij}$ and $\delta_{ij} \leq \beta_{ij}$. Since $\mathbf{A}$ and $\mathbf{B}$ are consistent, $\alpha_{ij} \leq \alpha_{jj}$ and $\beta_{ij} \leq \beta_{jj}$. Combining this result with the previous result yields $\delta_{ij} \leq \alpha_{jj}$ and $\delta_{ij} \leq \beta_{jj}$. This implies that $\delta_{ij} \leq \min(\alpha_{jj}, \beta_{jj})$, and thus $\delta_{ij} \leq \delta_{jj}$, for all i and j. Therefore, $\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$. $\square$

2.5 Message Logging and Checkpointing

A message is called *logged* if and only if its data and the index of the state interval that it started in its receiver process are *both* recorded on stable storage. The predicate *logged*(i, σ) is true if and only if the message that started state interval σ in process i is logged.

The predicate *checkpoint*(i, σ) is true if and only if there exists a checkpoint on stable storage that records the state of process i in state interval σ. When a process is created, it is immediately checkpointed before it begins execution, and thus, checkpoint$(i, 0)$ is true for all processes i.

For every state interval σ of some process, there must be *some* checkpoint on stable storage for that process with a state interval index no larger than σ,

since there is at least always a checkpoint on stable storage for state interval 0.

> **Definition 5** The *effective checkpoint* for a state interval σ of some process i is the checkpoint on stable storage for process i with the largest state interval index ϵ such that $\epsilon \leq \sigma$.

A state interval of a process is called *stable* if and only if all messages received by the process to start state intervals after its effective checkpoint are logged. The predicate $stable(i, \sigma)$ is true if and only if state interval σ of process i is stable.

> **Definition 6** If σ is the state interval index for some process i, and if ϵ is the state interval index of the effective checkpoint for state interval σ of process i, then state interval σ of process i is *stable* if and only if
>
> $$\forall \alpha, \epsilon < \alpha \leq \sigma \left[\text{logged}(i, \alpha) \right] .$$

Any stable state interval σ for a process can be recreated by restoring the process from the effective checkpoint (with state interval index ϵ) and replaying to it in order any logged messages to begin state intervals $\epsilon+1$ through σ.

The checkpoint of a process includes the complete current dependency vector for the process. Each logged message, though, only gives the single dependency created in the receiver by this message. The complete dependency vector for any stable state interval of some process is always known, though, since all messages that started state intervals since the effective checkpoint must be logged.

2.6 Recoverable System States

A system state is called *recoverable* if and only if all component process states are *stable* and the resulting system state is *consistent*. To recover the state of the system, it must be possible to recover the states of the component processes, and for this system state to be meaningful, it must be possible to have reached this state through normal execution of the system from its initial state.

> **Definition 7** If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathcal{S}$, $\mathbf{D}$ is *recoverable* if and only if
>
> $$\mathbf{D} \in \mathcal{C} \wedge \forall i \left[\text{stable}(i, \delta_{ii}) \right] .$$

Let the set $\mathcal{R} \subseteq \mathcal{S}$ be the set of *recoverable* system states that have occurred during any single execution of some system. Thus,

$$\mathcal{R} = \{ \mathbf{D} \in \mathcal{S} \mid \mathbf{D} \text{ is recoverable} \} .$$

Since only consistent system states can be recoverable, $\mathcal{R} \subseteq \mathcal{C} \subseteq \mathcal{S}$.

> **Lemma 2** The set $\mathcal{R}$ forms a sublattice of the system history lattice.

Proof For any $\mathbf{A}, \mathbf{B} \in \mathcal{R}$, $\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$ and $\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$, by Lemma 1. Since the state of each process in $\mathbf{A}$ and $\mathbf{B}$ is stable, all process states in $\mathbf{A} \cup \mathbf{B}$ and $\mathbf{A} \cap \mathbf{B}$ are stable as well. Thus, $\mathbf{A} \cup \mathbf{B} \in \mathcal{R}$ and $\mathbf{A} \cap \mathbf{B} \in \mathcal{R}$, and $\mathcal{R}$ forms a sublattice. □

2.7 The Current Recovery State

In recovering after a failure, we wish to restore the state of the system to the "most recent" recoverable state that is possible from the information available, in order to minimize the amount of reexecution necessary to complete the recovery. The system history lattice corresponds to this notion of time, and the following theorem establishes the existence of a *single* maximum recoverable state under this ordering.

> **Theorem 2** There is always a unique maximum recoverable system state in $\mathcal{S}$.

Proof $\mathcal{R} \subseteq \mathcal{S}$, and by Lemma 2, $\mathbf{A} \cup \mathbf{B} \in \mathcal{R}$ for any $\mathbf{A}, \mathbf{B} \in \mathcal{R}$. Since $\mathbf{A} \preceq \mathbf{A} \cup \mathbf{B}$ and $\mathbf{B} \preceq \mathbf{A} \cup \mathbf{B}$, the unique maximum in $\mathcal{S}$ is simply

$$\bigcup_{\mathbf{D} \in \mathcal{R}} \mathbf{D} ,$$

which must be unique since $\mathcal{R}$ forms a sublattice of the system history lattice. □

The maximum recoverable system state at any time is called its *current recovery state*. The following lemma shows that the current recovery state of the system never decreases.

> **Lemma 3** If the current recovery state of the system is $\mathbf{R} = [\rho_{**}]$, then, for each process i, the system can always be recovered without needing to roll back any state interval $\sigma \leq \rho_{ii}$.

Proof $\mathbf{R}$ will always remain consistent, and for each process i, state interval ρ_{ii} will always remain stable. Since $\mathcal{R}$ forms a sublattice, any new current recovery state established after $\mathbf{R}$ must be greater than $\mathbf{R}$ in the lattice. By Definition 1, this implies that the state interval index for each process in any new current recovery state must be greater than or equal to ρ_{ii}. Therefore, for each process i, no state interval $\sigma \leq \rho_{ii}$ will ever need to be rolled back. □

Corollary 1 If all messages received by executing processes are *eventually* logged, there is no possibility of the domino effect in the system.

Proof If all messages are eventually logged, all state intervals of all processes eventually become stable by Definition 6, and thus new recoverable states must become possible through Definition 7. By Lemma 3, these states will never need to be rolled back. □

2.8 Committing Output

If some state interval of a process must be rolled back to recover a consistent system state, any output messages sent while that state interval is being reexecuted after recovery may not be the same as those originally sent. Any processes that received such messages will be orphans and must also be rolled back to a point before these messages were received.

However, messages sent to the *outside world*, such as those to the user's display terminal, cannot be treated in the same way. Since the outside world generally cannot be rolled back, any messages sent to the outside world must be delayed until it is known that the state interval from which they were sent will never need to be rolled back, at which time they may be *committed* by releasing them. This theorem establishes when it is safe to commit an output message sent to the outside world.

Corollary 2 If the current recovery state of the system is $\mathbf{R} = [\rho_{**}]$, then any message sent by a process i from a state $\sigma \leq \rho_{ii}$ may be committed.

Proof Follows directly from Lemma 3. □

2.9 Garbage Collection

While the system is operating, checkpoints and logged messages accumulate on stable storage in case they are needed for some future recovery. This data may be removed from stable storage whenever doing so will not interfere with the ability of the system to recover as needed. The following two theorems establish when this can safely be done.

Corollary 3 Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state. For each process i, if ϵ_i is the state interval index of the effective checkpoint for its state interval ρ_{ii}, then any checkpoint for process i with state interval index $\sigma < \epsilon_i$ may be released from stable storage.

Proof Follows directly from Lemma 3. □

Corollary 4 Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state. For each process i, if ϵ_i is the state interval index of the effective checkpoint for its state interval ρ_{ii}, then any message that begins a state interval in process i with index $\sigma \leq \epsilon_i$ may be released from stable storage.

Proof Follows directly from Lemma 3. □

3 Recovery State Algorithm

3.1 Introduction

As the system executes, new logged messages and checkpoints arrive on stable storage. Occasionally, some combination of this information may define a new current recovery state by creating a new recoverable system state greater than the existing current recovery state. Theorem 2 of Section 2 established that there is always a unique maximum recoverable state at any time. Conceptually, this state may be found by an exhaustive search of all combinations of stable process state intervals until the maximum combination is found. However, such a search would be too expensive in practice, and an effective means of limiting this search space is important.

The *recovery state algorithm* monitors checkpoints and logged messages as they arrive on stable storage and decides if each allows an advance in the current recovery state of the system. The algorithm is invoked each time a process state interval becomes stable; it is incremental in that it only examines information that has changed since its last execution, rather than recomputing the entire current recovery state on each execution. Since it only uses information on stable storage, it can handle any number of concurrent process failures.

3.2 The Basic Algorithm

Each time some new state interval σ of some process k becomes stable, the algorithm attempts to form a new current recovery state in which the state of process k is advanced to state interval σ. It does so by including any state intervals from other processes that are necessary to make this new system state consistent. The check for consistency is performed by a direct application of the definition of system state consistency from Section 2. The algorithm succeeds if all such state intervals included are stable, making this

new consistent system state composed entirely of stable process state intervals. Otherwise, no new current recovery state is possible.

An outline of the basic recovery state algorithm is shown below. Some details are omitted from this outline for clarity; these will be discussed later after the basic algorithm is described. Let $\mathbf{R} = [\rho_{*\,*}]$ be the current recovery state of the system. When state interval σ of process k becomes stable, the following steps are taken by the algorithm:

1. If $\sigma \leq \rho_{k\,k}$, then exit the algorithm, since the current recovery state is already in advance of state interval σ of process k.

2. Make a new dependency matrix $\mathbf{D} = [\delta_{*\,*}]$ from $\mathbf{R}$, with row k replaced by the dependency vector for state interval σ of process k.

3. Loop on step 3 while $\mathbf{D}$ is not consistent. That is, loop while there exists some i and j for which $\delta_{i\,j} > \delta_{j\,j}$, which shows that some process i depends on a state interval of process j greater than process j's current state interval in $\mathbf{D}$.

   Find a stable state interval $\alpha \geq \delta_{i\,j}$ of process j. If state interval $\delta_{i\,j}$ is stable, let α be $\delta_{i\,j}$; otherwise, choose some later state interval of process j for α, if one exists:

   (a) If no such state interval exists for α that is stable, exit the algorithm, but remember to recheck this later.

   (b) Otherwise, replace row j of $\mathbf{D}$ with the dependency vector for state interval α of process j.

4. The dependency matrix $\mathbf{D}$ is now consistent and composed only of stable process state intervals. It is thus recoverable. Replace $\mathbf{R}$ with this new system state $\mathbf{D}$, making it the new current recovery state.

3.3 Some Details

Lemma 4 The state interval α chosen for process j during each iteration in step 3 must be the minimum $\alpha \geq \delta_{i\,j}$ that is stable.

Proof As a process executes, no element of its dependency vector can decrease. Thus, the dependencies of any state interval of process j after this minimum α will be at least as large as those of state interval α. Clearly, if state interval $\delta_{i\,j}$ is stable, its dependencies will be exactly only those that are necessary; any later state interval of process j may have additional dependencies that state interval $\delta_{i\,j}$ does not have. Using the minimum set of dependencies possible with the stable process states that are available will restrict the solutions the least. □

Lemma 5 The comparisons in step 3 to check if $\mathbf{D}$ is a consistent system state may be made in any order without affecting the final resulting dependency matrix.

Proof Since the only change made to $\mathbf{D}$ during the loop of step 3 is the replacement of row j with the dependency vector for state interval α, the only effect that the order of these comparisons has is the order in which these row replacements are performed.

First, each replacement of row j can only increase $\delta_{j\,j}$, since row j is only replaced when $\delta_{i\,j} > \delta_{j\,j}$, and the new dependency vector for that row is always chosen such that its state interval $\alpha \geq \delta_{i\,j} > \delta_{j\,j}$.

Second, any row replacements required by the replacement of some row j will still be required after the replacement of row j', $j' \neq j$, unless row j also required the replacement of row j', and the state interval index of the new row j' is greater than that required by row j, in which case this new row j' would still be required if row j's requirement had been met first.

Thus, the dependency vector left in each row of $\mathbf{D}$ when the algorithm terminates will always have the maximum state interval index of any vector placed in that row during the loop of step 3, regardless of the order that the row replacements are made. □

Lemma 6 When state interval σ of process k becomes stable, the basic algorithm finds *some* recoverable system state $\mathbf{D} = [\delta_{*\,*}]$ with $\delta_{k\,k} = \sigma$, if any such system state exists.

Proof Any system state found must be recoverable since only stable process state intervals are included by the algorithm, and the resulting system state is checked for consistency. As each row of $\mathbf{D}$ is replaced, the dependencies that must be satisfied grow as little as possible with the stable process states that are available, as shown in Lemma 4. Since the state interval for any process used in $\mathbf{D}$ never decreases as the algorithm executes, the state interval for process k in any recoverable state found will never be less than σ. If the algorithm finds that it needs any state interval of process k greater than σ, no recoverable state is possible, since the fact that state interval σ of process k is now stable has no effect on such a

system state, and any such recoverable state that exists would have already been found by some earlier execution of the algorithm □

Lemma 7 No stable process state interval that was deferred in step 3a needs to be rechecked until step 4 advances the current recovery state.

Proof Suppose some state interval σ of process k becomes stable and the algorithm determines that no new recoverable state is possible. By Lemma 6, this means that no consistent set of stable process state intervals $\mathbf{A} = [\alpha_{**}]$ is available with $\alpha_{kk} = \sigma$.

Now suppose some new state interval σ' of process k' becomes stable. If the algorithm determines that no new recoverable state is possible, there is no consistent set of stable process state intervals $\mathbf{B} = [\beta_{**}]$ available with $\beta_{k'k'} = \sigma'$. The only effect that this new state interval σ' of process k' can have on the earlier evaluation of state interval σ of process k is that some recoverable state may now be possible with the state interval index of process k' set to σ', but the algorithm has already determined that *no* such recoverable state is possible. Thus, there is no need to recheck any earlier deferred stable process states in this case. □

This lemma shows when it is necessary to recheck any deferred stable state intervals. It also gives a method to greatly limit the set of those deferred stable state intervals that need to be rechecked, rather than rechecking all such state intervals that are not yet included in the current recovery state.

Corollary 5 When the current recovery state advances from $\mathbf{R} = [\rho_{**}]$ to some new state $\mathbf{R}' = [\rho'_{**}]$, the stable process states that were deferred earlier by step 3a and should now be rechecked are those with a direct dependency on some state interval σ of any process i such that $\rho_{ii} < \sigma \leq \rho'_{ii}$.

Proof The proof follows directly from the proof of Lemma 7. □

3.4 Correctness

Theorem 3 The recovery state algorithm always finds the current recovery state of the system.

Proof First, by Lemma 6, the algorithm only finds recoverable system states. Also, any such system states found will be greater than the previous current recovery state since at least the new state interval σ for process k is always greater than the previous state interval index for process k in the current recovery state. Lemma 6 also shows that if some new recoverable state can be formed when state interval σ of process k becomes stable, the algorithm finds one. Lemma 7 shows when it is necessary to recheck any process state interval that could not be added to a new current recoverable state when it became stable, and Corollary 5 shows which state intervals should be rechecked then. By rechecking all those state intervals at the correct times, the maximum recoverable state must be found. □

3.5 An Efficient Procedure

The algorithm described in Sections 3.2 and 3.3 can be implemented efficiently by making some observations about the execution of the algorithm, based on Lemma 5.

When step 3 examines the dependency matrix $\mathbf{D}$ on each iteration, there may be many pairs of i and j for which $\delta_{ij} > \delta_{jj}$, indicating that several different rows of the matrix need to be replaced. These required row replacements can be entered into a list of pending replacements as each is discovered. Since initially only row k of $\mathbf{D}$ has been changed from the current recovery state, and since on each iteration, only one row is replaced at a time, only the single changed row needs to be compared against the diagonal elements of $\mathbf{D}$ for consistency. Then, only the diagonal elements of the matrix are needed during the execution of the algorithm. The list of pending row replacements only needs to remember the maximum index of any state interval needed for each process, since the dependency vector that the algorithm leaves in each row of $\mathbf{D}$ is the one for that process with the maximum state interval index, regardless of the order that the replacements are performed.

The function *FIND_RV*, shown in Figure 3, is a procedure to to implement steps 1 through 3 of the basic algorithm, taking advantage of these observations. This procedure finds a new recoverable state based on state interval σ of process k, if such a state exists. The list of pending row replacements is maintained in *NEED*, such that *NEED*[i] is always the maximum index of any state interval in process i that is currently needed to replace row i of the matrix. If no row replacements are currently needed for some process j, then *NEED*[j] is set to $\perp$. A vector, *RV*, is used instead of the full dependency matrix, where *RV*[i] is diagonal element i of the corresponding de-

```
function FIND_RV(RV, k, σ)
  if σ ≤ RV[k] then return true;
  for i ← 1 to n do NEED[i] ← ⊥;
  RV[k] ← σ;
  for i ← 1 to n do
    if DV_k^σ[i] > RV[i] then NEED[i] ← DV_k^σ[i];
  while ∃ i such that NEED[i] ≠ ⊥ do
    α ← minimum such that
      α ≥ NEED[i] and stable(i, α);
    if no such α then return false;
    RV[i] ← α;
    NEED[i] ← ⊥;
    for j ← 1 to n do
      if DV_i^α[j] > RV[j] then
        NEED[j] ← max(NEED[j], DV_i^α[j]);
return true;
```

Figure 3 Finding a new recoverable state

pendency matrix, which is also the state interval index of process i in the recoverable state. As each row is replaced, only the corresponding single element of RV is changed.

Using function *FIND_RV*, the full recovery state algorithm can now be stated. This algorithm, shown in Figure 4, initially calls *FIND_RV* on the state interval that just became stable. If no new recoverable state is found, the algorithm exits since no change in the current recovery state is possible from this new stable state. If *FIND_RV* returns success, the result becomes the new current recovery state, and the algorithm checks if any other recoverable states greater than this result can now exist. The sets $DEFER_j^\beta$ keep track of those deferred stable process state intervals that should be rechecked when the current recovery state advances over state interval β of process j. The set *WORK* keeps a list of those deferred states that are to be rechecked by the algorithm because the current recovery state has been advanced.

```
WORK ← { (k, σ) };
while WORK ≠ ∅ do
  remove some state (x, θ) from WORK;
  if θ > CRS[x] then
    for j ← 1 to n do
      NEWCRS[j] ← CRS[j];
    if FIND_RV(NEWCRS, x, θ) = true
      then
        for j ← 1 to n do
          for β ← CRS[j] + 1 to NEWCRS[j] do
            WORK ← WORK ∪ DEFER_j^β;
            DEFER_j^β ← ∅;
          CRS[j] ← NEWCRS[j];
      else
        for j ← 1 to n do
          β ← DV_x^θ[j];
          if β > CRS[j] then
            DEFER_j^β ← DEFER_j^β ∪ { (x, θ) };
```

Figure 4 The recovery state algorithm

4 Related Work

A number of fault-tolerance recovery methods based on message logging and checkpointing have been published in the literature. This includes ones using pessimistic logging protocols such as Auros [1], Publishing [6], and sender-based message logging [3], as well as optimistic methods [9]. The model and recovery state algorithm presented in Sections 2 and 3 can be applied to each of these and used to reason about their correctness.

Our model is more general than is required by recovery methods based on pessimistic message logging, but the definitions of consistency, stability, and recoverability still apply, and the recovery state algorithm still computes the correct current recovery state. In this case, the current recovery state is identical to the state of the system at the time the failure occurred, since orphan processes are not possible. Since message logging is synchronous, however, a simpler recovery state algorithm is possible that takes advantage of the order that information arrives on stable storage. In particular, checkpoints never add new information for the algorithm, since messages are always logged in ascending order by the index of the state interval that they start in their receivers, and all messages received before a checkpoint have already been logged before the checkpoint can be recorded.

Recovery based on optimistic logging protocols requires the full generality of our model, however. Since orphan processes are possible when using optimistic logging, recovery from a failure is more difficult. Any orphan processes must be rolled back during recovery to achieve a consistent state. Since there is no synchronization between message logging, checkpointing, and computation, information for the recovery state algorithm may arrive on stable storage at any time and in any order. Thus, the algorithm must be able to make use of all this information in order to advance

the current recovery state to its *maximum* possible value at all times.

Our model and algorithm differ in several ways from those used for optimistic recovery by Strom and Yemini [9]. First, Strom and Yemini require reliable delivery of messages between processes. As a result, their definition of consistency differs from ours by requiring all messages sent to have been received. Our model does not require reliable delivery, but it can be incorporated easily by inserting a return acknowledgement message immediately following each message receipt, with our definition of consistency remaining unchanged. Second, although their system checkpoints processes in order to shorten recovery times and release old logged messages from stable storage, they do not take advantage of these checkpoints in computing the current maximum recoverable state in their system. Our algorithm uses both checkpoints and logged messages to compute the maximum recoverable state and thus may find recoverable states that their algorithm does not. Finally, our algorithm requires only the current state interval index of the sending process to be carried in each message, and requires only a vector of *direct* dependencies to be maintained by each process. In contrast, their method requires each process to maintain a vector of its *transitive* dependencies, and requires each message to be tagged with this vector, which has size linear in the number of processes. This added complexity does allow control of recovery in their system to be more decentralized than in ours.

5 Conclusion

From a performance standpoint, optimistic message logging protocols appear to be desirable. They seem to constitute the right performance tradeoff in operating environments where failures are rare and failure-free performance is of primary concern. The recovery state algorithm of Section 3 represents an improvement on earlier work with recovery based on optimistic message logging by Strom and Yemini [9]. Although their algorithm eventually achieves a recoverable state, this state may not be optimal. Furthermore, their methods require reliable communication and seem more complex than the method presented here.

This work unifies existing approaches to fault tolerance based on message logging and checkpointing published in the literature, including those using pessimistic message logging methods [1, 6, 3] and those using optimistic methods [9]. By using this model to reason about these types of fault-tolerance recovery methods, properties that are independent of the message logging protocol used can be deduced and proven. We have shown that there is always a unique maximum recoverable system state, which never decreases, and that in a system where all messages received are eventually logged, the domino effect cannot occur. The use of this general model allows more attention to be paid instead to designing efficient message logging protocols.

Acknowledgements

We would like to thank Rick Bubenik, John Carter, Matthias Felleisen, Gerald Fowler, and Elaine Hill for many helpful discussions on this material and for their comments on earlier drafts of this paper.

References

[1] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99, ACM, October 1983.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19, IEEE Computer Society, June 1987.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Xerox Palo Alto Research Center, Palo Alto, California, April 1979.

[6] Michael L. Powell and David L. Presotto. Publishing: a reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109, ACM, October 1983.

[7] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[8] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[9] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.