JOURNAL OF ALGORITHMS 11, 462–491 (1990)

Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing*

DAVID B. JOHNSON AND WILLY ZWAENEPOEL

Department of Computer Science, Rice University, P.O. Box 1892, Houston, Texas 77251-1892

Received September 12, 1989

Message logging and checkpointing can provide fault tolerance in distributed systems in which all process communication is through messages. This paper presents a general model for reasoning about recovery in these systems. Using this model, we prove that the set of recoverable system states that have occurred during any single execution of the system forms a lattice, and that therefore, there is always a *unique* maximum recoverable system state, which never decreases. Based on this model, we present an algorithm for determining this maximum recoverable state and prove its correctness. Our algorithm utilizes all logged messages *and* checkpoints, and thus always finds the maximum recoverable state possible. Previous recovery methods using optimistic message logging and checkpointing have not considered the existing checkpoints, and thus may not find this maximum state. Furthermore, by utilizing the checkpoints, some messages received by a process before it was checkpointed may not need to be logged. Using our algorithm also adds less communication overhead to the system than do previous methods. Our model and algorithm can be used with any message logging protocol, whether pessimistic or optimistic, but their full generality is only required with optimistic logging protocols.

1. INTRODUCTION

Message logging and checkpointing can be used to provide fault tolerance in a distributed system in which all process communication is through messages. Each message received by a process is saved in a *message log* on

*This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under Contract ONR N00014-88-K-0140.

0196-6774/90 $3.00

stable storage [11, 1], and the state of each process is occasionally saved as a *checkpoint* on stable storage. No coordination is required between the checkpointing of different processes or between message logging and checkpointing. The execution of each process is assumed to be deterministic between received messages, and all processes are assumed to execute on fail-stop processors [17].

Typically, these systems use a *pessimistic* protocol for message logging. Each message is *synchronously* logged as it is received, either by blocking the receiver until the message is logged [2, 14], or by blocking the receiver if it attempts to send a new message before all received messages are logged [8]. Recovery based on pessimistic message logging is straightforward. A failed process is reloaded from its most recent checkpoint, and all messages originally received by the process after this checkpoint was written are replayed to it from the log in the same order in which they were received before the failure. Using these messages, the process then reexecutes to the state it had after originally receiving them. Messages sent by the process during this reexecution that are duplicates of those sent before the failure are ignored.

Other systems, though, use an *optimistic* message logging protocol [21, 19]. The receiver of a message is not blocked, and messages are logged *asynchronously* after receipt, for example by grouping several messages and writing them to stable storage in a single operation. However, the current state of a process can only be recovered if all messages received by the process since it was last checkpointed have been logged. Because other processes may *depend on* states that cannot be recovered after a failure, recovery using optimistic message logging is more difficult than with pessimistic logging. These dependencies between processes arise through communication in the system, since any part of the state of a process may be included in a message. When a process receives a message, the current state of that process then depends on the state of the sender from which the message was sent.

A process that has received a message form some failed process that was sent from a more recent state than its latest state that can be recovered becomes an *orphan* process at the time of the failure. During recovery, each orphan process must be rolled back to a state before the message that caused it to become an orphan was received. Rolling back this process may cause other processes to become orphans, which must also be rolled back during recovery. The *domino effect* [15, 16] is an uncontrolled propagation of such process rollbacks and must be avoided to guarantee progress in the system in spite of failures. Recovery based on optimistic message logging must construct the "most recent" combination of process states that be recovered such that no process is an orphan. Since optimistic logging protocols avoid synchronization delays during

message logging, they can outperform pessimistic logging protocols in the absence of failures. Although the recovery procedure required with optimistic logging protocols is also more complex than with pessimistic protocols, it is only used when a failure occurs.

This paper presents a general model for reasoning about distributed systems using message logging and checkpointing to provide fault tolerance. With this model, we prove that the set of recoverable system states that have occurred during any single execution of the system forms a lattice, and that therefore, there is always a *unique* maximum recoverable system state, which never decreases. Based on this model, we present an algorithm for determining this unique maximum recoverable system state and prove its correctness. Our algorithm always finds this maximum recoverable system state, by utilizing all logged messages *and* checkpoints. Previous fault-tolerance methods using optimistic message logging and checkpointing [21, 19] have not considered the existing checkpoints and thus may not find this maximum state. Furthermore, by utilizing checkpoints, some messages received by a process before its checkpoint was recorded may not need to be logged. The use of our algorithm also adds less communication overhead to the system than do these other optimistic methods. Our model and algorithm can be used with any message logging protocol, whether pessimistic or optimistic, but their full generality is only required with optimistic logging protocols.

Section 2 of this paper presents our model for reasoning about these systems, and Section 3 describes our algorithm for finding the maximum recoverable system state. Using this algorithm to recover from a failure in the system is discussed in Section 4. Section 5 relates this work to other message logging and checkpointing methods, and Section 6 summarizes the contributions of this work.

2. The Model

This section presents a general model for reasoning about the behavior and correctness of recovery methods using message logging and checkpointing. The model is based on the dependencies between the states of processes that result from communication in the system. The state of each process is represented by its dependencies, and the state of the system is represented by a collection of process states. The model does not assume the use of any particular message logging protocol and applies equally well to systems using either pessimistic or optimistic message logging methods. All processes are assumed to execute on fail-stop processors [17] connected by a communication network, but reliable delivery of messages on the network is not required.

2.1. *Process States*

The execution of each process is divided into separate intervals by the messages that the process receives. Each interval, called a *state interval* of the process, is a *deterministic* sequence of execution, started by the receipt of the next message by the process. The execution of a process within a single state interval is completely determined by the state of the process at the time that the message is received and by the contents of the message. A process may *send* any number of messages to other processes during any state interval.

Within a process, each state interval of that process is uniquely identified by a sequential *state interval index*, which is simply a count of messages received by the process. Processes may be dynamically created and destroyed, but each process must be identified by a globally unique process identifier. Logically, these identifiers are assumed to be in the range 1 through n for a system of n processes. The creation of a process is modeled by its receipt of message number 0, and process termination is modeled by its receipt of one final message following the sequence of real messages received by the process. All messages *sent* by a process are tagged by its current state interval index.

When a process i receives a message sent by some process j, the state of process i then depends on the state that process j had at the time that the message was sent. The state of a process is represented by its current set of dependencies on all other processes. For each process i, these dependencies are represented by a *dependency vector*

$$\langle \delta_* \rangle = \langle \delta_1, \delta_2, \delta_3, \ldots, \delta_n \rangle,$$

where n is the total number of processes in the system. Component j of process i's dependency vector, δ_j, is set to the *maximum* index of any state interval of process j on which process i currently depends. If process i has no dependency on any state interval of process j, then δ_j, is set to $\perp$, which is less than all possible state interval indices. Component i of process i's own dependency vector is always set to the index of process i's current state interval. The dependency vector of a process names only those state intervals on which the process *directly* depends, resulting from the receipt of a message sent from that state interval in the sending process. Only the maximum index of any state interval of each other process on which this process depends is recorded, since the execution of a process within each state interval is deterministic, and since this state interval naturally also depends on all previous intervals of the same process.

Processes cooperate to maintain their dependency vectors by tagging all messages sent with the current state interval index of the sending process, and by remembering in each process the maximum index tagging any

message received from each other process. During any single execution of the system, the current dependency vector of any process is uniquely determined by the state interval index of that process. No component of the dependency vector of any process can decrease through failure-free execution of the system.

2.2. *System States*

A system state is a collection of process states, one for each process in the system. These process states need not all have existed in the system at the same time. A system state is said to have *occurred* during some execution of the system if all component process states have each individually occurred during this execution. A system state is represented by an $n \times n$ *dependency matrix*

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} \delta_{11} & \delta_{12} & \delta_{13} & \cdots & \delta_{1n} \\ \delta_{21} & \delta_{22} & \delta_{23} & \cdots & \delta_{2n} \\ \delta_{31} & \delta_{32} & \delta_{33} & \cdots & \delta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_{n1} & \delta_{n2} & \delta_{n3} & \cdots & \delta_{nn} \end{bmatrix},$$

where row i, δ_{ij}, $1 \le j \le n$, is the dependency vector for the state of process i included in this system state. Since for all i, component i of process i's dependency vector is always the index of its current state interval, the diagonal of the dependency matrix, δ_{ii}, $1 \le i \le n$, is always set to the current state interval index of each process contained in the system state.

Let $\mathcal{S}$ be the set of all system states that have occurred during any *single* execution of some system. The *system history relation*, $\prec$, is a partial order on the set $\mathcal{S}$, such that one system state precedes another in this relation if and only if it *must* have occurred first during this execution. The relation $\prec$ can be expressed in terms of the state interval index of each process shown in the dependency matrices representing these system states.

DEFINITION 2.1. If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then

$$\mathbf{A} \preccurlyeq \mathbf{B} \Leftrightarrow \forall i[\alpha_{ii} \le \beta_{ii}],$$

and

$$\mathbf{A} \prec \mathbf{B} \Leftrightarrow (\mathbf{A} \preccurlyeq \mathbf{B}) \wedge (\mathbf{A} \neq \mathbf{B}).$$

The system history relation differs from Lamport's *happened before* relation [10] in that it orders the system states that result from events

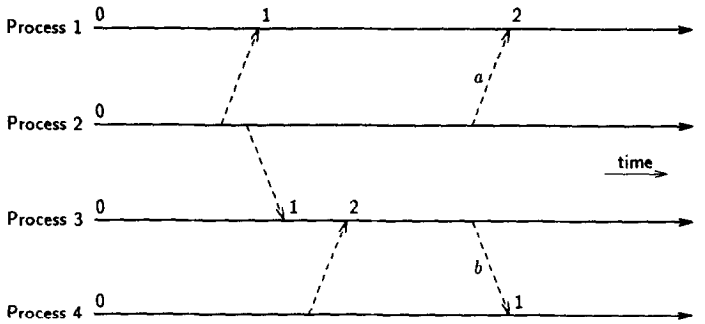

FIG. 1. The system history partial order. Neither message a nor message b must have been received first.

rather than the events themselves and that only state intervals (started by the receipt of a message) constitute events.

To illustrate this partial order, Fig. 1 shows a system of four communicating processes. The horizontal lines represent the execution of each process, with time progressing from left to right. Each arrow between processes represents a message sent from one process to another, and the number at each arrow gives the index of the state interval started by the receipt of that message. The last message received by process 1 is message a, and the last message received by process 4 is message b. Consider the two possible system states **A** and **B**, such that in state **A**, message a has been received but message b has not, and in state **B**, message b has been received but message a has not. These two system states can be represented by the dependency matrices

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

System states **A** and **B** are incomparable under the system history relation. This is shown by a comparison of the circled values on the diagonals of these two dependency matrices. In the execution of the system, neither state **A** nor state **B** *must* have occurred first, because neither message a nor message b must have been received first.

2.3. *The System History Lattice*

A system state describes the set of messages that have been received by each process. For any two system states **A** and **B** in $\mathcal{S}$, the *meet* of **A** and

B, written $\mathbf{A} \sqcap \mathbf{B}$, represents a system state that has also occurred during this execution of the system, in which each process has received only those messages that it has received in *both* **A** and **B**. This can be expressed in terms of the dependency matrices representing these two system states by copying each row from the corresponding row of one of the two original matrices, depending on which matrix has the *smaller* entry on its diagonal in that row.

DEFINITION 2.2. If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathscr{S}$, the *meet* of **A** and **B** is $\mathbf{A} \sqcap \mathbf{B} = [\phi_{**}]$, such that

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

Likewise, for any two system states **A** and **B** in $\mathscr{S}$, the *join* of **A** and **B**, written $\mathbf{A} \sqcup \mathbf{B}$, represents a system state that has also occurred during this execution of the system, in which each process has received only those messages that it has received in *either* **A** or **B**. This can be expressed in terms of the dependency matrices representing these two system states by copying each row from the corresponding row of one of the two original matrices, depending on which matrix has the *larger* entry on its diagonal in that row.

DEFINITION 2.3. If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathscr{S}$, the *join* of **A** and **B** is $\mathbf{A} \sqcup \mathbf{B} = [\theta_{**}]$, such that

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

Continuing the example of Section 2.2 illustrated in Fig. 1, the meet and join of states **A** and **B** can be represented by the dependency matrices

$$\mathbf{A} \sqcap \mathbf{B} = \begin{bmatrix} 1 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & \perp & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{A} \sqcup \mathbf{B} = \begin{bmatrix} 2 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & 2 & 1 \end{bmatrix}.$$

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 2.1. *The set $\mathscr{S}$, ordered by the system history relation, forms a lattice. For any* $\mathbf{A}, \mathbf{B} \in \mathscr{S}$, *the greatest lower bound of* **A** *and* **B** *is* $\mathbf{A} \sqcap \mathbf{B}$, *and the least upper bound of* **A** *and* **B** *is* $\mathbf{A} \sqcup \mathbf{B}$.

Proof. Follows directly from the construction of system state meet and join in Definitions 2.2 and 2.3. □

2.4. *Consistent System States*

Because the process states composing a system state need not all have existed at the same time, some system states may represent an impossible state of the system. A system state is called *consistent* if it *could* have been seen at some instant by an outside observer during the preceding execution of the system from its initial state, regardless of the relative speeds of the component processes [4]. After recovery from a failure, the system must be recovered to a consistent system state. This ensures that the total execution of the system is equivalent to *some* possible failure-free execution.

In this model, since all process communication is through messages and since processes execute deterministically between received messages, a system state is consistent if no component process has received a message that has not been sent yet in this system state and that cannot be sent through the future deterministic execution of the sender. Since process execution is only deterministic within each state interval, this is true only if no process has received a message that will not be sent before the end of the sender's current state interval contained in this system state. Any messages shown by a 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 **D**. Then suppose process i sent some message n (such as an acknowledgment of message m), which could show the receipt of m. If message n has been received in system state **D**, state **D** will be inconsistent because message n (not message m) is shown to have been received but not yet sent. If message n has not been received yet in state **D**, no effect of either message can be seen in **D**, and **D** is therefore still consistent.

The definition of a consistent system state can be expressed in terms of the dependency matrices representing system states. If a system state is consistent, then for each process i, no other process j depends on a state interval of process i beyond process i's current state interval. In the dependency matrix, for each column i, no element in column i in any row j is larger than the element on the diagonal of the matrix in column i (and row i), which is process i's current state interval index.

DEFINITION 2.4. If $\mathbf{D} = [\delta * *]$ is some system state in $\mathscr{S}$, **D** is *consistent* if and only if

$$\forall i, j\left[\delta_{ji} \leq \delta_{ii}\right].$$

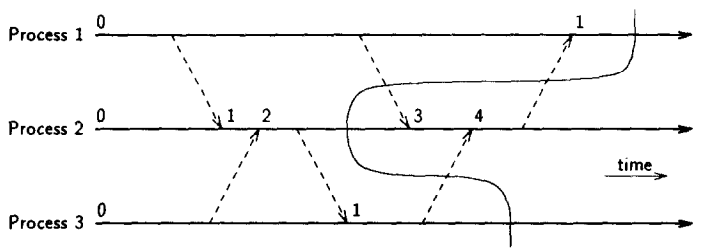

FIG. 2. An inconsistent system state.

For example, consider the system of three processes whose execution is shown in Fig. 2. The state of each process has been observed where the curve intersects the line representing the execution of that process, and the resulting system state is represented by the dependency matrix

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

This system state is not consistent, since process 1 has received a message (to begin state interval 1) from process 2, which was sent beyond the end of process 2's current state interval. This message has not been sent yet by process 2 and cannot be sent by process 2 through its future deterministic execution. In terms of the dependency matrix shown above, since δ_{12} is greater than δ_{22}, the system state represented by this matrix is not consistent.

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

$$\mathscr{C} = \{\mathbf{D} \in \mathscr{S} \mid \mathbf{D} \text{ is consistent}\}.$$

THEOREM 2.2. *The set $\mathscr{C}$, ordered by the system history relation, forms a sublattice of the system history lattice.*

Proof. Let $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ be system states in $\mathscr{C}$. By Definition 2.4, since $\mathbf{A} \in \mathscr{C}$ and $\mathbf{B} \in \mathscr{C}$, $\alpha_{ji} \leq \alpha_{ii}$ and $\beta_{ji} \leq \beta_{ii}$, for all i and j. It suffices to show that $\mathbf{A} \sqcap \mathbf{B} \in \mathscr{C}$ and $\mathbf{A} \sqcup \mathbf{B} \in \mathscr{C}$.

Let $\mathbf{A} \sqcap \mathbf{B} = [\phi_{**}]$. By Definition 2.2, and because $\mathbf{A}$ and $\mathbf{B}$ both occurred during the same execution of the system and no element in the dependency vector of any process can decrease through execution of the process, then $\phi_{ji} = \min(\alpha_{ji}, \beta_{ji})$, for all i and j. Thus, $\phi_{ji} \leq \alpha_{ji}$ and $\phi_{ji} \leq \beta_{ji}$, for all i and j. Since $\mathbf{A} \in \mathscr{C}$ and $\mathbf{B} \in \mathscr{C}$, $\phi_{ji} \leq \alpha_{ji} \leq \alpha_{ii}$ and $\phi_{ji} \leq \beta_{ji} \leq \beta_{ii}$. Thus, $\phi_{ji} \leq \min(\alpha_{ii}, \beta_{ii})$, and $\phi_{ji} \leq \phi_{ii}$, for all i and j. Therefore, $\mathbf{A} \sqcap \mathbf{B} \in \mathscr{C}$.

Let $\mathbf{A} \sqcup \mathbf{B} = [\theta_{**}]$. By Definition 2.3, $\theta_{ji} = \alpha_{ji}$ or $\theta_{ji} = \beta_{ji}$, and $\theta_{ii} = \max(\alpha_{ii}, \beta_{ii})$, for all i and j. Since $\mathbf{A} \in \mathscr{E}$ and $\mathbf{B} \in \mathscr{E}$, $\theta_{ji} \leq \theta_{ii}$ for all i and j as well. Therefore, $\mathbf{A} \sqcup \mathbf{B} \in \mathscr{E}$. □

2.5. *Message Logging and Checkpointing*

As the system executes, messages are recorded on stable storage in a message log. A message is called *logged* if and only if its data and the index of the state interval that it started in the process that received it are *both* recorded on stable storage. Logged messages remain on stable storage until no longer needed for recovery from any possible future failure of the system (Section 2.9). The predicate *logged*(i, σ) is true if and only if the message that started state interval σ of process i is logged.

When a process is created, its initial state is saved on stable storage as a checkpoint (in state interval 0). Each process is also independently checkpointed at times during its execution. Each checkpoint remains on stable storage until no longer needed for recovery from any possible future failure of the system (Section 2.9). For every state interval σ of each process, there must then be *some* checkpoint of that process on stable storage with a state interval index no larger than σ.

DEFINITION 2.5. 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 $\varepsilon \leq \sigma$.

A state interval of a process is called *stable* if and only if it can be recreated on demand from information currently on stable storage. This is true if and only if all received messages that started state intervals in the process after its state interval recorded in the effective checkpoint are logged. The predicate *stable*(i, σ) is true if and only if state interval σ of process i is stable.

DEFINITION 2.6. State interval σ of process i is *stable* if and only if

$$\forall \alpha, \varepsilon < \alpha \leq \sigma[\text{logged}(i, \alpha)],$$

where ε is the index of the state interval of process i recorded in the effective checkpoint for state interval σ.

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

The checkpointing of a process need not be coordinated with the logging of messages received by that process. In particular, a process may be checkpointed at any time, and the state interval recorded in that

checkpoint is then stable, regardless of whether all previous messages received by that process have been logged. Thus, if a state interval σ of some process i is stable and its effective checkpoint records its state interval ε, then all state intervals α of process i, $\varepsilon \leq \alpha \leq \sigma$, must be stable, but some state intervals $\beta < \varepsilon$ of process i may not be stable.

Each checkpoint of a process includes the complete current dependency vector of the process. Each logged message only contains the state interval index of the sending process at the time that the message was sent (tagging the message), but the complete dependency vector for any stable state interval of any process is always known, since all messages that started state intervals after the effective checkpoint must be logged.

2.6. *Recoverable System States*

A system state is called *recoverable* if and only if all component process state intervals are *stable* and the resulting system state is *consistent*. That is, to recover the state of the system, it must be possible to recreate the states of the component processes, and for this system state to be meaningful, it must be possible to have occurred through failure-free execution of the system from its initial state.

DEFINITION 2.7. If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathscr{S}$, $\mathbf{D}$ is *recoverable* if and only if

$$\mathbf{D} \in \mathscr{C} \wedge \forall i\left[\text{stable}(i, \delta_{ii})\right].$$

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

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

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

THEOREM 2.3. *The set $\mathscr{R}$, ordered by the system history relation, forms a sublattice of the system history lattice.*

Proof. For any $\mathbf{A}, \mathbf{B} \in \mathscr{R}$, $\mathbf{A} \sqcap \mathbf{B} \in \mathscr{C}$ and $\mathbf{A} \sqcup \mathbf{B} \in \mathscr{C}$, by Theorem 2.2. Since the state interval of each process in $\mathbf{A}$ and $\mathbf{B}$ is stable, all process state intervals in $\mathbf{A} \sqcap \mathbf{B}$ and $\mathbf{A} \sqcup \mathbf{B}$ are stable as well. Thus, $\mathbf{A} \sqcap \mathbf{B} \in \mathscr{R}$ and $\mathbf{A} \sqcup \mathbf{B} \in \mathscr{R}$, and $\mathscr{R}$ forms a sublattice. □

2.7. *The Current Recovery State*

During recovery, the state of the system is restored 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 recov-

ery. The system history lattice corresponds to this notion of time, and the following theorem establishes the existence of a *single* maximum recoverable system state under this ordering.

THEOREM 2.4. *There is always a unique maximum recoverable system state in* $\mathscr{S}$.

Proof. The unique maximum in $\mathscr{S}$ is simply

$$\bigsqcup_{\mathbf{D} \in \mathscr{R}} \mathbf{D},$$

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

DEFINITION 2.8. At any time, the *current recovery state* of the system is the state to which the system will be restored if any failure occurs in the system at that time.

In this model, the *current recovery state* of the system is always the unique maximum system state that is currently recoverable.

LEMMA 2.1. *During any single execution of the system, the current recovery state never decreases.*

Proof. Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state of the system at some time. Dependencies can only be added to state **R** by the receipt of a new message, which would cause the receiving process to begin a new state interval, resulting in a new system state. Thus, system state **R** itself must remain consistent. Since logged messages and checkpoints are not removed until no longer needed, state interval ρ_{ii} for each process i must remain stable until no longer needed. Thus system state **R** itself must remain recoverable. Since the set $\mathscr{R}$ forms a lattice, any new current recovery state established after state **R** must be greater than **R**. □

As discussed in Section 1, the *domino effect* [15, 16] is an uncontrolled propagation of rollbacks necessary to recover the system state following a failure. In this model, an occurrence of the domino effect would take the form of a propagation of dependencies that prevent the current recovery state from advancing. The following lemma establishes a sufficient condition for preventing the domino effect.

LEMMA 2.2. *If all messages received by each process are* eventually *logged, there is no possibility of the domino effect in the system.*

Proof. Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state of the system at some time. For all state intervals σ of each process k, $\sigma > \rho_{kk}$, if all messages are eventually logged, state interval σ of process k will eventu-

ally become stable, by Definition 2.6. By Lemma 2.1, the current recovery state never decreases, and thus, by Definition 2.7, new system states $\mathbf{R}'$, $\mathbf{R} \prec \mathbf{R}'$, must eventually become recoverable and become the new current recovery state. The domino effect is thus avoided, since the current recovery state eventually increases. □

2.8. *The Outside World*

During execution, processes may interact with the *outside world*, which consists of everything outside the system itself. Examples of interactions with the outside world include receiving input from a human user and writing information on the user's display terminal. All interactions with the outside world are modeled as messages either received from the outside world or sent to the outside world. Messages from the outside world received by a process must be logged in the same way as other messages received by a process.

Messages sent to the outside world, though, cannot be treated in the same way as those sent to other processes within the system, since messages to the outside world may cause irreversible side effects. To guarantee that the state of the outside world is consistent with the state of the system restored during recovery, any message sent to the outside world must be delayed until it is known that the state interval from which it was sent will never be rolled back. It can then be *committed* by releasing it to the outside world. The following lemma establishes when it is safe to commit a message sent to the outside world.

LEMMA 2.3. *If the current recovery state of the system is* $\mathbf{R} = [\rho_{**}]$, *then any message sent by a process* i *from a stable interval* $\sigma \leq \rho_{ii}$ *may be committed.*

Proof. Follows directly from Lemma 2.1 and Definition 2.1. □

2.9. *Garbage Collection*

During operation of the system, checkpoints and logged messages must remain on stable storage until they are no longer needed for any possible future recovery of the system. They may be removed from stable storage only whenever doing so will not interfere with the ability of the system to recover as needed. The following two lemmas establish when this can be done safely.

LEMMA 2.4. *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 state interval* ρ_{ii} *of process* i, *then any checkpoint of process* i *with state interval*

index $\sigma < \varepsilon_i$ cannot be needed for any future recovery of the system and may be removed from stable storage.

Proof. Follows directly from Lemma 2.1 and Definitions 2.1 and 2.5. □

LEMMA 2.5. *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 state interval ρ_{ii} of process i, then any message that begins a state interval $\sigma \leq \varepsilon_i$ in process i cannot be needed for any future recovery of the system and may be removed from stable storage.*

Proof. Follows directly from Lemma 2.1 and Definitions 2.1 and 2.5. □

3. THE RECOVERY STATE ALGORITHM

Theorem 2.4 shows that in any system using message logging and checkpointing to provide fault tolerance, there is always a unique maximum recoverable system state. This maximum state is the *current recovery state*, the state to which the system will be restored following a failure. At any time, the current recovery state could be found by an exhaustive search, over all combinations of currently stable process state intervals, for the maximum consistent combination, but such a search would be too expensive in practice. Our *recovery state algorithm* finds the current recovery state more efficiently.

The recovery state algorithm is invoked once for each process state interval that becomes stable, either because a new checkpoint has recorded the process in that state interval or because all messages received since the effective checkpoint for that interval are now logged. It use the dependency vectors of these stable process state intervals to form new dependency matrices that represent consistent system states, which are therefore also recoverable. It is a centralized algorithm, using this information collected from the execution of the system. Since all process state intervals considered by the algorithm are stable, all information used by the algorithm has been recorded on stable storage. The algorithm is therefore restartable and can handle any number of concurrent process failures, including a total failure. The algorithm is incremental in that it uses the existing known maximum recoverable system state and advances it when possible, based on the fact that a new process state interval has become stable.

For each new state interval σ of some process k that becomes stable, the algorithm determines if a new current recovery state exists. It first

attempts to find *some* new recoverable system state in which the state of process k has advanced to state interval σ. If no such system state exists, the current recovery state of the system has not changed. The algorithm records the index of this state interval and its process identifier on one or more lists to be checked again later. If a new recoverable system state is found, the algorithm searches for other greater recoverable system states, using the appropriate lists. The new current recovery state is the maximum recoverable system state found in this search.

3.1. *Finding a New Recoverable System State*

The heart of the recovery state algorithm is the procedure *FIND_REC*. Given *any* recoverable system state $\mathbf{R} = [\rho_{**}]$ and some stable state interval σ of some process k with $\sigma > \rho_{kk}$, *FIND_REC* attempts to find a new recoverable system state in which the state of process k is advanced at least to state interval σ. It does this by also including any stable state interval from other processes that are necessary to make the new system state consistent, applying the definition of a consistent system state in Definition 2.4. The procedure succeeds if such a consistent system state can be composed from the set of process state intervals that are currently stable. Since the state of process k has advanced, the new recoverable system state found must be greater than state **R** in the system history lattice.

Input to the procedure *FIND_REC* consists of the dependency matrix of some recoverable system state $\mathbf{R} = [\rho_{**}]$, the process identifier k and state interval index $\sigma > \rho_{kk}$ of a stable state interval of process k, and the dependency vector for each stable process state interval θ of each process x such that $\theta > \rho_{xx}$. Conceptually, *FIND_REC* performs the following steps:

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

2. Loop on step 2 while **D** is not consistent. That is, loop while there exists some i and j for which $\delta_{ji} > \delta_{ii}$. This shows that state interval δ_{jj} of process j depends on state interval δ_{ji} of process i, which is greater than process i's current state interval δ_{ii} in **D**.

Find the minimum index α of any *stable* state interval of process i such that $\alpha \geq \delta_{ji}$:

   (a) If no such state interval α exists, exit the algorithm and return **false**.

   (b) Otherwise, replace row i of **D** with the dependency vector for this state interval α of process i.

```
function FIND_REC(RV, k, σ)

    RV[k] ← σ;
    for i ← 1 to n do MAX[i] ← max(RV[i], DV_k^σ[i]);

    while ∃ i such that MAX[i] > RV[i] do
        α ← minimum index such that
            α ≥ MAX[i] ∧ stable(i, α);
        if no such state interval α exists then return false;
        RV[i] ← α;
        for j ← 1 to n do MAX[j] ← max(MAX[j], DV_i^α[j]);

    return true;
```

FIG. 3. Procedure to find a new recoverable state.

3. The system state represented by **D** is now consistent and is composed entirely of stable process state intervals. It is thus recoverable and greater than **R**. Return **true**.

An efficient implementation of procedure *FIND_REC* is shown in Fig. 3. This implementation operates on a vector *RV*, rather than on the full dependency matrix representing the system state. For all i, $RV[i]$ contains the diagonal element from row i of the corresponding dependency matrix. When *FIND_REC* is called, each $RV[i]$ contains the state interval index of process i in the given recoverable system state. The dependency vector of each stable state interval θ of process x is represented by the vector DV_x^θ. As each row of the matrix is replaced in the outline above, the corresponding single element of *RV* is changed in *FIND_REC*. Also, the maximum element from each column of the matrix is maintained in the vector *MAX*, such that for all i, $MAX[i]$ contains the maximum element in column i of the corresponding matrix.

LEMMA 3.1. *If function FIND_REC is called with a known recoverable system state* $\mathbf{R} = [\rho_{**}]$ *and state interval* σ *of process* k *such that* $\sigma > \rho_{kk}$, *FIND_REC returns* **true** *if there exists* some *recoverable system state* $\mathbf{R}' = [\rho'_{**}]$, *such that* $\mathbf{R} \prec \mathbf{R}'$ *and* $\rho'_{kk} \geq \sigma$, *and returns* **false** *otherwise. If FIND_REC returns* **true**, *then on return,* $RV[i] = \rho'_{ii}$, *for all* i.

Proof. The predicate of the **while** loop determines whether the dependency matrix corresponding to *RV* and *MAX* is consistent, by Definition 2.4. When the condition becomes false and the loop terminates, the matrix must be consistent because, in each column i, no element is larger than the element on the diagonal in that column. Thus, if *FIND_REC* returns **true**, the system state returned in *RV* must be consistent. This system state must also be recoverable, since its initial component process state

intervals are stable and only stable process state intervals are used to replace its entries during the execution of *FIND_REC*.

The following loop invariant is maintained by function *FIND_REC* at the top of the **while** loop on each iteration:

> If a recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists such that $\mathbf{R} \prec \mathbf{R}'$ and $\rho'_{ii} \geq RV[i]$, for all i, then $\rho'_{ii} \geq MAX[i]$.

The invariant must hold initially because any consistent state must have $RV[i] \geq MAX[i]$, for all i, and any state $\mathbf{R}'$ found such that $\rho'_{ii} \geq RV[i]$ must then have $\rho'_{ii} \geq RV[i] \geq MAX[i]$. On each subsequent iteration of the loop, the invariant is maintained by choosing the *smallest* index $\alpha \geq MAX[i]$ such that state interval α of process i is stable. For the matrix to be consistent, α must not be less than $MAX[i]$. By choosing the minimum such α, all components of DV_i^α are also minimized because no component of the dependency vector can decrease through execution of the process. Thus, after replacing row i of the matrix with DV_i^α, the components of *MAX* are minimized, and for any recoverable (consistent) state $\mathbf{R}'$ that exists, the condition $\rho'_{ii} \geq MAX[i]$ must still hold for all i.

If no such state interval $\alpha \geq MAX[i]$ of process i is currently stable, then no recoverable system state $\mathbf{R}'$ can exist, since any such $\mathbf{R}'$ must have $\rho'_{ii} \geq MAX[i]$. This is exactly the condition under which the procedure *FIND_REC* returns **false**. □

Suppose state interval σ of process k depends on state interval δ of process i, then procedure *FIND_REC* searches for the minimum $\alpha \geq \delta$ that is the index of a state interval of process i that is currently stable. For the set of process state intervals that are currently stable, the dependency on state interval δ of process i has been *transferred* to state interval α of process i (including the case of $\alpha = \delta$), and state interval σ of process k is said to currently have a *transferred dependency* on state interval α of process i.

DEFINITION 3.1. A state interval σ of some process k, with dependency vector $\langle \delta_* \rangle$, has a *transferred dependency* on a state interval α of process i if and only if:

(1) $\alpha \geq \delta_i$,

(2) state interval α of process i is stable, and

(3) there does not exist another stable state interval β of process i such that $\alpha > \beta \geq \delta_i$.

The transitive closure of the transferred dependency relation from state interval σ of process k describes the set of process state intervals that *may* be used in any iteration of the **while** loop of procedure *FIND_REC*,

```
if σ ≤ CRS[k] then exit;

NEWCRS ← CRS;

if ¬FIND_REC(NEWCRS, k, σ) then
   for i ← 1 to n do if i ≠ k then
      β ← DV_k^σ[i];
      if β > CRS[i] then DEFER_i^β ← DEFER_i^β ∪ {(k, σ)};
   exit;

WORK ← DEFER_k^σ;
β ← σ − 1;
while ¬stable(k, β) do
      WORK ← WORK ∪ DEFER_k^β; β ← β − 1;

while WORK ≠ ∅ do
      remove some (x, θ) from WORK;
      if θ > NEWCRS[x] then
         RV ← NEWCRS;
         if FIND_REC(RV, x, θ) then NEWCRS ← RV;
      if θ ≤ NEWCRS[x] then
         WORK ← WORK ∪ DEFER_x^θ;
         β ← θ − 1;
         while ¬stable(x, β) do
               WORK ← WORK ∪ DEFER_x^β; β ← β − 1;

CRS ← NEWCRS;
```

FIG. 4. The recovery state algorithm, invoked when state interval σ of process k becomes stable.

when invoked for this state interval. Although only a subset of these state intervals will actually be used, the exact subset used in any execution depends on the order in which the **while** loop finds the next i that satisfies the predicate.

3.2. *The Complete Algorithm*

Using function *FIND_REC*, the complete recovery state algorithm can now be stated. The algorithm, shown in Fig. 4, uses a vector *CRS* to record the state interval index of each process in the current recovery state of the system. When a process is created, its entry in *CRS* is initialized to 0. When some state interval σ of some process k becomes stable, if this state interval is in advance of the old current recovery state in *CRS*, the algorithm checks if a new current recovery state exists. During the execution, the vector *NEWCRS* is used to store the maximum known recoverable system state, which is copied back to *CRS* at the completion of the algorithm.

When invoked, the algorithm calls *FIND_REC* with the old current recovery state and the identification of the new stable process state

interval. The old current recovery state is the maximum known recoverable system state, and the new stable state interval is interval σ of process k. If *FIND_REC* returns **false**, then no greater recoverable system state exists in which the state of process k has advanced at least to state interval σ. Thus, the current recovery state of the system has not changed, as shown by the following two lemmas.

LEMMA 3.2. *When state interval σ of process k becomes stable, if the current recovery state changes from $\mathbf{R} = [\rho_{**}]$ to $\mathbf{R}' = [\rho'_{**}]$, $\mathbf{R} \prec \mathbf{R}'$, then $\rho'_{kk} = \sigma$.*

Proof. By contradiction. Suppose the new current recovery state $\mathbf{R}'$ has $\rho'_{kk} \neq \sigma$. Only state interval σ of process k has become stable, since $\mathbf{R}$ was the current recovery state, but process k in the new current recovery state $\mathbf{R}'$ is not in state interval σ. Thus, all process state intervals in $\mathbf{R}'$ must have been stable before state interval σ of process k became stable. Therefore, system state $\mathbf{R}'$ must have been recoverable before state interval σ of process k became stable. Since $\mathbf{R} \prec \mathbf{R}'$, then $\mathbf{R}'$ must have been the current recovery state before state interval σ of process k became stable, contradicting the assumption that $\mathbf{R}$ was the original current recovery state. Thus, if the current recovery state has changed, then $\rho'_{kk} = \sigma$. □

LEMMA 3.3. *When state interval σ of process k becomes stable, if the initial call to FIND_REC by the recovery state algorithm returns* **false**, *then the current recovery state of the system has not changed.*

Proof. By Lemma 3.2, if the current recovery state changes from $\mathbf{R} = [\rho_{**}]$ to $\mathbf{R}' = [\rho'_{**}]$ when state interval σ of process k becomes stable, then $\rho'_{kk} = \sigma$. However, a **false** return from *FIND_REC* indicates that *no* recoverable system state $\mathbf{R}'$ exists with $\rho'_{kk} \geq \sigma$, such that $\mathbf{R} \prec \mathbf{R}'$. Therefore, the current recovery state cannot have changed. □

Associated with each state interval β of each process i that is in advance of the known current recovery state is a set $DEFER_i^\beta$, which records the identification of any stable process state intervals that depend on state interval β of process i. That is, if the current recovery state of the system is $\mathbf{R} = [\rho_{**}]$, then for all i and β such that $\beta > \rho_{ii}$, $DEFER_i^\beta$ records the set of stable process state intervals that have β in component i of their dependency vector. All *DEFER* sets are initialized to the empty set when the corresponding process is created. If *FIND_REC* returns **false** when some new process state interval becomes stable, that state interval is entered in at least one *DEFER* set. The algorithm uses these sets to limit its search space for the new current recovery state.

If the initial call to *FIND_REC* by the recovery state algorithm returns **true**, a new greater recoverable system state has been found. Additional calls to *FIND_REC* are used to search for any other recoverable system states that exist that are greater than the one returned by the last call to *FIND_REC*. The new current recovery state of the system is the state returned by the last call to *FIND_REC* that returned **true**. The algorithm uses a result of the following lemma to limit the number of calls to *FIND_REC* required.

LEMMA 3.4. *Let* $\mathbf{R} = [\rho_{**}]$ *be the existing current recovery state of the system, and then let state interval* σ *of process k become stable. For any stable state interval* θ *of any process x such that* $\theta > \rho_{xx}$, *no recoverable system state* $\mathbf{R}' = [\rho'_{**}]$ *exists with* $\rho'_{xx} \geq \theta$ *if state interval* θ *of process x does not depend on state interval* σ *of process k by the transitive closure of the transferred dependency relation.*

Proof. Since state interval θ of process x is in advance of the old current recovery state, it could not be made part of any recoverable system state $\mathbf{R}'$ before state interval σ of process k became stable. If it does not depend on state interval σ of process k by the transitive closure of the transferred dependency relation, then the fact that state interval σ has become stable cannot affect this.

Let δ be the maximum index of any state interval of process k that state interval θ of process x is related to by this transitive closure. Clearly, any new recoverable system state $\mathbf{R}' \neq \mathbf{R}$ that now exists with $\rho'_{xx} \geq \theta$ must have $\rho'_{kk} \geq \delta$, by Definitions 3.1 and 2.4, and since no component of any dependency vector decreases through execution of the process. If $\delta > \sigma$, then system state $\mathbf{R}'$ was recoverable before state interval σ became stable, contradicting the assumption that $\theta > \rho_{kk}$. Likewise, if $\delta < \sigma$, then $\mathbf{R}'$ cannot exist now if it did not exist before state interval σ of process k became stable, since state interval δ must have been stable before state interval σ became stable. Since both cases lead to a contradiction, no such recoverable system state $\mathbf{R}'$ can now exist without this dependency through the transitive closure. □

The **while** loop of the recovery state algorithm uses the *DEFER* sets to traverse the transitive closure of the transferred dependency relation backward from state interval σ of process k. Each state interval θ of some process x visited on this traversal depends on state interval σ of process k by this transitive closure. That is, either state interval θ of process x has a transferred dependency on state interval σ of process k, or it has a transferred dependency on some other process state interval that depends on interval σ of process k by this transitive closure. The traversal uses the set *WORK* to record those process state intervals from which the traversal

must still be performed. When *WORK* has been emptied, the new current recovery state has been found and is copied back to *CRS*.

During this traversal, any dependency along which no more **true** results from *FIND_REC* can be obtained is not traversed further. If the state interval θ of process x that is being considered is in advance of the maximum known recoverable system state, *FIND_REC* is called to search for a new greater recoverable system state in which process x has advanced at least to state interval θ. If no such recoverable system state exists, the traversal from this state interval is not continued, since *FIND_REC* will return **false** for all other state intervals that depend on state interval θ of process x by this transitive closure.

LEMMA 3.5. *If state interval β of process i depends on state interval θ of process x by the transitive closure of the transferred dependency relation, and if no recoverable system state $\mathbf{R} = [\rho_{**}]$ exists with $\rho_{xx} \geq \theta$, then no recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists with $\rho'_{ii} \geq \beta$.*

Proof. This follows directly from the definition of a transferred dependency in Definition 3.1. Either state interval β of process i has a transferred dependency on state interval θ of process x, or it has a transferred dependency on some other process state interval that depends on state interval θ of process x by this transitive closure. By this dependency, any such recoverable system state $\mathbf{R}'$ that exists must also have $\rho'_{xx} \geq \theta$, but no such recoverable system state exists, since $\mathbf{R}$ does not exist. Therefore, $\mathbf{R}'$ cannot exist. □

THEOREM 3.1. *If the recovery state algorithm is executed each time any state interval σ of any process k becomes stable, it will complete each time with $CRS[i] = \rho'_{ii}$, for all i, where $\mathbf{R}' = [\rho'_{**}]$ is the new current recovery state of the system.*

Proof. The theorem holds before the system begins execution, since $CRS[i]$ is initialized to 0 when each process i is created. Likewise, if any new process i is created during execution of the system, it is correctly added to the current recovery state by setting $CRS[i] = 0$.

When some state interval σ of some process k becomes stable, if the initial call to *FIND_REC* returns **false**, the current recovery state remains unchanged, by Lemma 3.3. In this case, the recovery state algorithm correctly leaves *CRS* unchanged.

If this call to *FIND_REC* returns **true** instead, the current recovery state has advanced as a result of this new state interval becoming stable. Let $\mathbf{R} = [\rho_{**}]$ be the old current recovery state before state interval σ of process k became stable, and let $\mathbf{D} = [\delta_{**}]$ be the system state returned by this call to *FIND_REC*. Then $\mathbf{R} \prec \mathbf{D}$, by Lemma 3.1. Although the

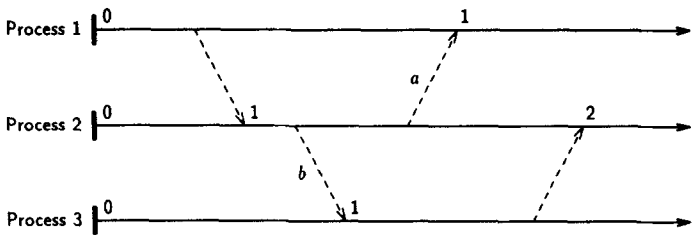

FIG. 5. An example system execution.

system state **D** may be less than the new current recovery state **R'**, $\mathbf{D} \preccurlyeq \mathbf{R}'$ because the set of recoverable system states forms a lattice.

The **while** loop of the recovery state algorithm finds the new current recovery state by searching forward in the lattice of recoverable system states, without backtracking. This search is performed by traversing backward through the transitive closure of the transferred dependency relation, using the information in the *DEFER* sets. For each state interval θ of each process x examined by this loop, if no recoverable system state exists in which the state of process x has advanced at least to state interval θ, the traversal from this state interval is not continued. By Lemmas 3.4 and 3.5, this loop considers all stable process state intervals for which a new recoverable system state can exist. Thus, at the completion of this loop, the traversal has been completed, and the last recoverable system state found must be new current recovery state. The algorithm finally copies this state from *NEWCRS* to *CRS*. □

3.3. *An Example*

Figure 5 shows the execution of a system of three processes. Each process has been checkpointed in its state interval 0, but no other checkpoints have been written. Also, a total of four messages have been received in the system, but no messages have been logged yet. Thus, only state interval 0 for each process is stable, and the current recovery state of the system is composed of state interval 0 of each process. In the recovery state algorithm, $CRS = \langle 0, 0, 0 \rangle$, and all *DEFER* sets are empty.

If message a from process 2 to process 1 now becomes logged, state interval 1 of process 1 becomes stable and has a dependency vector of $\langle 1, 1, \perp \rangle$. The recovery state algorithm is executed and calls *FIND_REC* with $\sigma = 1$ and $k = 1$ for state interval 1 of process 1. *FIND_REC* sets *RV* to $\langle 1, 0, 0 \rangle$ and *MAX* to $\langle 1, 1, 0 \rangle$. Since $MAX[2] > RV[2]$, a stable state interval $\alpha \geq 1$ of process 2 is needed to make a consistent system state. However, no such state interval of process 2 is currently stable, and

FIND_REC therefore returns **false**. The recovery state algorithm changes $DEFER_2^1$ to $\{(1, 1)\}$ and exits, leaving *CRS* unchanged at $\langle 0, 0, 0 \rangle$.

Next, if process 2 is checkpointed in state interval 2, this state interval becomes stable. Its dependency vector is $\langle 0, 2, 1 \rangle$. The recovery state algorithm calls *FIND_REC*, which sets *RV* to $\langle 0, 2, 0 \rangle$ and *MAX* to $\langle 0, 2, 1 \rangle$. Since no state interval $\alpha \geq 1$ of process 3 is stable, *FIND_REC* returns **false**. The recovery state algorithm sets $DEFER_3^1$ to $\{(2, 2)\}$ and exits, leaving *CRS* unchanged again.

Finally, if message *b* from process 2 to process 3 becomes logged, state interval 1 of process 3 becomes stable, and has a dependency vector of $\langle \perp, 1, 1 \rangle$. The recovery state algorithm calls *FIND_REC*, which sets *RV* to $\langle 0, 0, 1 \rangle$ and *MAX* to $\langle 0, 1, 1 \rangle$. Since $MAX[2] > RV[2]$, a stable state interval $\alpha \geq 1$ of process 2 is required. State interval 2 of process 2 is the minimum such stable state interval. Using its dependency vector, *RV* and *MAX* are updated, yielding the value $\langle 0, 2, 1 \rangle$ for both. This sytem state is consistent, and *FIND_REC* returns **true**. The maximum known recoverable system state in *NEWCRS* has then been increased to $\langle 0, 2, 1 \rangle$.

The *WORK* set is initialized to $DEFER_3^1 = \{(2, 2)\}$, and the **while** loop of the algorithm begins. When state interval 2 of process 2 is checked, it is not in advance of *NEWCRS*, so the call to *FIND_REC* is skipped. The sets $DEFER_2^2$ and $DEFER_2^1$ are added to *WORK*, making $WORK = \{(1, 1)\}$. State interval 1 of process 1 is then checked by the **while** loop. Procedure *FIND_REC* is called, which sets both *RV* and *MAX* to $\langle 1, 2, 1 \rangle$, and therefore returns **true**. The maximum known recoverable system state in *NEWCRS* is updated by this call to $\langle 1, 2, 1 \rangle$. The set $DEFER_1^1$ is added to *WORK*, but since $DEFER_1^1 = \emptyset$, this leaves *WORK* empty. The **while** loop then terminates, and the value left in $NEWCRS = \langle 1, 2, 1 \rangle$ is copied back to *CRS*. The system state represented by this value of *CRS* is the new current recovery state of the system.

This example illustrates a unique feature of our recovery state algorithm. Our algorithm uses both logged messages and checkpoints in its search for the maximum recoverable system state. Although only two of the four messages received during this execution of the system have been logged, the current recovery state has advanced due to the checkpoint of process 2. In fact, the two remaining unlogged messages need never be logged, since the current recovery state has advanced beyond their receipt.

4. Failure Recovery

The recovery state algorithm can be used in recovering from any number of process failures in the system, including a total failure of all processes. Before beginning recovery, the state of any surviving processes

and any surviving messages that have been received but not yet logged may be used to further advance the current recovery state. This surviving information is volatile and has not been included in the computation of the current recovery state, since the current recovery state reflects only information that has been recorded on stable storage. Thus, the state of each process that did not fail must be written to stable storage as an additional checkpoint of that process, and all received messages that remain after the failure that have not yet been logged must be logged on stable storage. After the recovery state algorithm has been executed for each process state interval that becomes stable as a result of this, the current recovery state will be the maximum possible recoverable system state including this additional information that survived the failure.

To restore the state of the system to the current recovery state, the states of all failed processes must be restored, and any *orphan* processes must also be rolled back. Each failed process is restored by restarting it from the effective checkpoint for its state interval in the current recovery state, and then replaying to it from the log any messages received since that checkpoint was recorded. Using these logged messages, the recovering process deterministically reexecutes to restore its state to the state interval for this process in the current recovery state. Any other process currently executing in a state interval beyond the state interval of that process in the current recovery state is an *orphan*. To complete recovery, each orphan process is forced to fail and is restored to its state interval in the current recovery state in the same way as other failed processes. If additional processes fail during this recovery, the recovery may be restarted, since all information used is recorded on stable storage.

5. Related Work

5.1. *Optimistic Message Logging Methods*

Two other methods to support fault tolerance using optimistic message logging and checkpointing have been published in the literature. Our work has been partially motivated by Strom and Yemini's Optimistic Recovery [21], and recently Sistla and Welch have proposed a new optimistic message logging method [19], based in part on some aspects of both Strom and Yemini's system and our work. Our system is unique among these in that it always finds the *maximum* recoverable system state. Although these other systems occasionally checkpoint processes as our system does, they do not consider the existing checkpoints in finding the current recovery state. Our algorithm includes both checkpoints and logged messages in this search, and thus may find recoverable system states that these other

algorithms do not. Also, these other systems assume reliable delivery of messages on the network, using a channel between each pair of processes that does not lose or reorder messages. Thus, in their definitions of a consistent system state, Strom and Yemini require all messages sent to have been received, and Sistla and Welch require the sequence of messages received on each channel to be a prefix of those sent on it. Since our model does not assume reliable delivery, it can be applied to common real distributed systems that do not guarantee reliable delivery, such as those based on an Ethernet network. If needed, reliable delivery can also be incorporated into our model simply by assuming an acknowledgment message immediately following each message receipt.

In Strom and Yemini's Optimistic Recovery [21], each message sent is tagged with a *transitive* dependency vector, which has size proportional to the number of processes. Also, each process is required to locally maintain its knowledge of the message logging progress of each other process in a *log vector*, which is either periodically broadcast by each process or appended to each message sent. Our system tags each message only with the current state interval index of the sender. Information equivalent to the log vector is maintained by the recovery state algorithm, but uses no additional communication beyond that already required to log each message. Although communication of the transitive dependency vector and the log vector allows control of recovery to be less centralized and may result in faster commitment of output to the outside world, this additional communication may add significantly to the failure-free overhead of the system. Optimistic Recovery also includes an *incarnation number* as part of each state interval index to identify the number of times that the process has rolled back. This preserves the uniqueness of state interval indices across recoveries and allows recovery of different processes to proceed without synchronization. With our model, processes must synchronize during recovery to be notified of the reuse of the indices of any rolled back state intervals.

Sistla and Welch have proposed two alternative recovery algorithms based on optimistic message logging [19]. One algorithm tags each message sent with a transitive dependency vector as in Strom and Yemini's system, whereas the other algorithm tags each message only with the sender's current state interval index as in our system. To find the current recovery state, each process sends information about its message logging progress to all other processes, after which their second algorithm also exchanges additional messages, essentially to distribute the complete transitive dependency information. Each process then locally performs the same computation to find the current recovery state. This results in $O(n^2)$ messages for the first algorithm, and $O(n^3)$ messages for the second, where n is the number of processes in the system. In contrast, our algorithm requires no

additional communication beyond that necessary to log each message on stable storage. Again, this additional communication in their system allows control of recovery to be less centralized than in ours. However, the current recovery state must be frequently determined, so that output to the outside world can be committed quickly. Therefore, the increased communication in Sistla and Welch's algorithms may add substantial failure-free overhead to the system.

5.2. *Pessimistic Message Logging Methods*

Our system is more general than that required when using a pessimistic message logging protocol, but our model can still be applied and our recovery state algorithm correctly finds the maximum recoverable system state. A simpler algorithm, though, can be used to find the current recovery state when using a pessimistic logging protocol. The current recovery state is always composed of the most recent stable state interval of each process in the system, since the protocol prevents the system from entering any state in which the system state composed in this way is not consistent. In the protocols used by the TARGON/32 system [3], it predecessor Auros [2], and the Publishing mechanism [14], the receiver of a message is blocked until the message is logged, and therefore, each state interval is stable before the process begins execution in that state interval. In the sender-based message logging protocol [8], each process is instead blocked if attempts to send a new message when any messages it has received are not yet logged. This prevents any process from receiving a message sent from a state interval of the sender that is not yet stable and thus ensures that this system state is consistent. Optimistic message logging removes the need for synchronization between execution and message logging, and thus optimistic methods should outperform pessimistic methods when failures are infrequent.

5.3. *Other Methods*

The general approach used by these message logging and checkpointing methods has been called the *state machine approach* [18], which assumes that program execution for each input is deterministic and is based only on the program state at the time of the input and on the input itself. This approach is also used by the Time Warp system [6], through its Virtual Time method [7], using message logging and checkpointing. However, Virtual Time is designed to support the synchronization required by particular distributed applications such as discrete event simulation, rather than to provide general-purpose process fault tolerance.

Checkpointing has also been used without message logging to provide fault tolerance in distributed systems [4, 9]. A global checkpoint, composed

of an independent checkpoint for each process in the system, is recorded such that this set of checkpoints forms a consistent system state. The system can therefore be recovered by restoring each process to its state in any global checkpoint. This removes the need to log all messages received in the system, but to commit output to the outside world, global checkpointing must be performed frequently, which may substantially degrade the failure-free performance of the system. Also, process execution may be blocked during checkpointing in order to guarantee the recording of a consistent system state [9]. Message logging removes any need for synchronization during checkpointing and allows checkpointing to be performed less frequently without sacrificing the ability to commit output to the outside world.

Different forms of logging and checkpointing have also been used to support recovery in systems based on atomic transactions [12, 13, 20, 5]. Logging on stable storage is used to record state changes of modified objects during the execution of a transaction. Typically, the entire state of each object is recorded, although *logical logging* [1] records only the names of operations performed and their parameters, such that they can be reexecuted during recovery, much the same as reexecuting processes based on logged messages. Logging may proceed asynchronously during the execution of the transaction, but must be forced to stable storage before the transaction can commit. This is similar to the operation of optimistic message logging and the requirement that the system state must be recoverable before output may be committed to the outside world. Before the transaction can commit, additional synchronous logging is also required to ensure the atomicity of the commit protocol, which is not necessary with message logging and checkpointing methods. However, this extra logging can be reduced through the use of special commit protocols, such as the *Presumed Commit* and *Presumed Abort* protocols [12].

To recover a transaction using this logging, however, the entire transaction must be reexecuted, which may lengthen recovery times and may prevent the recovery of transactions whose running times exceed the mean time between failures in the system. Smaller transactions may be used to avoid these problems, but this increases the amount of logging and the frequency of stable storage synchronization. The QuickSilver system [5] addresses these problems by allowing individual transactions to be checkpointed during their execution. This avoids the need to entirely reexecute a transaction during recovery, but this transaction checkpoint must record a consistent state of all processes involved in the transaction, much the same as a global checkpoint in checkpointing systems without message logging. Recording this consistent transaction checkpoint may significantly delay the execution of the transaction, due to the synchronization needed to record a consistent state.

6. Conclusion

Optimistic message logging allows messages to be logged asynchronously, without blocking process execution. This improves failure-free performance of the system over pessimistic message logging methods, but requires a more complex recovery procedure. Optimistic message logging methods thus constitute a beneficial performance trade-off in environments where failures are infrequent and failure-free performance is of primary concern.

The recovery state algorithm and recovery procedure presented in this paper improve on earlier work with fault-tolerance using optimistic message logging by Strom and Yemini [21] and by Sistla and Welch [19]. Although their methods allow less centralized control of recovery and may allow output to the outside would to be committed earlier, they add significantly more communication to the system. Also, although these two systems checkpoint processes as in our system, they do not consider these existing checkpoints in determining the current recovery state of the system. Our algorithm considers both checkpoints and logged messages and thus may find recoverable system states that these other systems do not find. We have proven, based on our model of Section 2, that our algorithm always finds the *maximum* possible recoverable system state. Furthermore, by utilizing these checkpointed states, some messages received by a process before it was checkpointed may not need to be logged, as demonstrated by the example in Section 3.3.

This work unifies existing approaches to fault tolerance using message logging and checkpointing published in the literature, including those using pessimistic message logging [2, 14, 3, 8] and those using optimistic methods [21, 19]. By using this model to reason about these types of fault-tolerance methods, properties of them that are independent of the message logging protocol used can be deduced and proven. We have shown 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. There is thus always a unique maximum recoverable system state.

Acknowledgments

We thank Rick Bubenik, John Carter, Matthias Felleisen, Jerry Fowler, Pete Keleher, and Alejandro Schaffer for many helpful discussions on this material and for their comments on earlier drafts of this paper. The comments of the referees also helped to improve the clarity of the presentation.

REFERENCES

1. P. A. BERNSTEIN, V. HADZILACOS, AND N. GOODMAN, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, MA, 1987.
2. A. BORG, J. BAUMBACH, AND S. GLAZER, A message system supporting fault tolerance, *in* "Proceedings, Ninth ACM Symposium on Operating Systems Principles," pp. 90–99, October 1983.
3. A. BORG, W. BLAU, W. GRAETSCH, F. HERRMANN, AND W. OBERLE, Fault tolerance under UNIX, *ACM Trans. Comput. Systems*, **7**, No. 1 (1989), 1–24.
4. K. M. CHANDY AND L. LAMPORT, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Systems* **3**, No. 1 (1985), 63–75.
5. R. HASKIN, Y. MALACHI, W. SAWDON, AND G. CHAN, Recovery management in QuickSilver, *ACM Trans. Comput. Systems* **6**, No. 1 (1988), 82–108.
6. D. JEFFERSON, B. BECKMAN, F. WIELAND, L. BLUME, M. DILORETO, P. HONTALAS, P. LAROCHE, K. STURDEVANT, J. TUPMAN, V. WARREN, J. WEDEL, H. YOUNGER, AND S. BELLENOT, Distributed simulation and the Time Warp operating system, *in* "Proceedings, Eleventh ACM Symposium on Operating Systems Principles," pp. 77–93, November 1987.
7. D. R. JEFFERSON, Virtual Time, *ACM Trans. Programming Lang. Systems* **7**, No. 3 (1985), 404–425.
8. D. B. JOHNSON AND W. ZWAENEPOEL, Sender-based message logging, *in* "The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers," pp. 14–19, IEEE Computer Society, June 1987.
9. R. KOO AND S. TOUEG, Checkpointing and rollback-recovery for distributed systems, *IEEE Trans. Software Eng.* **SE-13**, No. 1 (1987), 23–31.
10. L. LAMPORT, Time, clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**, No. 7 (1978), 558–565.
11. B. W. LAMPSON AND H. E. STURGIS, "Crash Recovery in a Distributed Data Storage System," technical report, Xerox Palo Alto Research Center, Palo Alto, California, April 1979.
12. C. MOHAN, B. LINDSAY, AND R. OBERMARCK, Transaction management in the *R** distributed database management system, *ACM Trans. Database Systems* **11**, No. 4 (1986), 378–396.
13. B. M. OKI, B. H. LISKOV, AND R. W. SCHEIFLER, Reliable object storage to support atomic actions, *in* "Proceedings, Tenth ACM Symposium on Operating Systems Principles," pp. 147–159, December 1985.
14. M. L. POWELL AND D. L. PRESOTTO, Publishing: A reliable broadcast communication mechanism, *in* "Proceedings, Ninth ACM Symposium on Operating Systems Principles," pp. 100–109, October 1983.
15. B. RANDELL, System structure for software fault tolerance, *IEEE Trans. Software Eng.* **SE-1**, No. 2 (1975), 220–232.
16. D. L. RUSSELL, State restoration in systems of communicating processes, *IEEE Trans. Software Eng.* **SE-6**, No. 2 (1980), 183–194.
17. R. D. SCHLICHTING AND F. B. SCHNEIDER, Fail-stop processors: An approach to designing fault-tolerant distributed computing systems, *ACM Trans. Comput. Systems* **1**, No. 3 (1983), 222–238.
18. F. B. SCHNEIDER, "The State Machine Approach: A Tutorial," Technical Report TR 86-800, Cornell University, Ithaca, New York, June 1987; to appear in "Proceedings, Workshop on Fault-Tolerant Distributed Computing," Lecture Notes in Computer Science series, Springer-Verlag, New York.

19. A. P. SISTLA AND J. L. WELCH, Efficient distributed recovery using message logging, *in* "Proceedings, Eighth Annual ACM Symposium on Principles of Distributed Computing," August 1989.
20. A. Z. SPECTOR, Distributed transaction processing and the Camelot system, *in* "Distributed Operating Systems: Theory and Practice," Vol. 28 (Y. Paker, J.-P. Banatre, and M. Bozyiğit, Eds), NATO Advanced Science Institute Series F: Computer and Systems Sciences, pp. 331–353, Springler-Verlag, Berlin, 1987; Technical Report CMU-CS-87-100, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, January 1987.
21. R. E. STROM AND S. YEMINI, Optimistic recovery in distributed systems, *ACM Trans. Comput. Systems* **3**, No. 3 (1985), 204–226.