This paper was submitted to the 9th Symposium on Reliable Distributed Software (October 1990).

Finding the Maximum Recoverable System State in Optimistic Rollback Recovery Methods

David B. Johnson
Peter J. Keleher
Willy Zwaenepoel

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

(713) 527-4834

dbj@rice.edu

Abstract

In a distributed system using *rollback recovery*, information saved on stable storage during failure-free execution allows certain states of each process to be recovered after a failure. For example, in a deterministic system using message logging and checkpointing, a process state can be recovered only if all messages received by the process since its previous checkpoint have been logged. In a nondeterministic system using checkpointing alone, a process state can be recovered only if it has been recorded in a checkpoint. *Optimistic* rollback recovery methods in general record this information asynchronously, assuming that a suitable *recoverable* system state can be constructed for use during recovery. A system state is called *recoverable* if and only if it is *consistent* and the state of each individual process in that system state can be recovered.

This paper shows that in any system using optimistic rollback recovery, there is always a *unique* maximum recoverable system state, extending our previous result for systems using message logging and checkpointing. We also present a simple new algorithm for finding the maximum recoverable system state, and describe some experience with its implementation. These results can be applied both to deterministic and to nondeterministic systems. (KR) ←

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.

1 Introduction

In a distributed system using *rollback recovery*, information saved on stable storage during failure-free execution allows certain states of each process to be recovered after a failure. For example, in a system using message logging and checkpointing [Powell83, Johnson87, Borg89, Strom85, Strom88, Johnson88, Sistla89], in which all process execution between received messages is assumed to be deterministic, a process state can be recovered only if all messages received by the process since its previous checkpoint have been logged. In a system not assuming deterministic process execution, and using checkpointing alone to provide fault tolerance [Koo87, Chandy85, Bhargava88], a process state can be recovered only if it has been recorded in a checkpoint.

After a failure, the system must be restored to a *consistent* system state. Essentially, a system state is consistent if it *could* have occurred during the preceding execution of the system from its initial state, regardless of the relative speeds of individual processes [Chandy85]. This ensures that the total execution of the system is equivalent to *some* possible failure-free execution. To be able to recover a system state, all of its individual process states must be able to be recovered. A consistent system state in which each process state can be recovered is thus called a *recoverable* system state. *Optimistic* rollback recovery methods [Strom85, Strom88, Johnson88, Sistla89, Bhargava88] in general record the recovery information asynchronously, assuming that a suitable *recoverable* system state can be constructed for use during recovery.

This paper shows that in any system using optimistic rollback recovery, there is always a *unique* maximum recoverable system state, extending our previous result for systems using message logging and checkpointing [Johnson88]. This paper also presents a simple new algorithm for finding the maximum recoverable system state at any time in a system using optimistic rollback recovery, and describes some experience with the implementation of this algorithm. These results can be applied both to systems in which all process execution between received messages is deterministic, and to systems in which no such assumption is made.

In Section 2 of this paper, the system model assumed by this work is presented, including formal definitions of *consistent* and *recoverable* system states. Section 3 presents our algorithm for finding the maximum recoverable system state, and Section 4 describes our implementation experience. Related work is discussed in Section 5, and in Section 6, we present conclusions.

2 System Model

The model presented in this section is an extension of our model for reasoning about systems using message logging and checkpointing [Johnson88, Johnson89]. It is based on the notion of *dependency* between the states of processes that results from communication between those processes. This section summarizes the model and describes its new features, and establishes the existence of a single *unique* maximum recoverable system state in any system using optimistic rollback recovery.

Each system is defined to be either *deterministic* or *nondeterministic*. In a *deterministic* system, the execution of each process is assumed to be deterministic between received messages. That is, after a process receives a message, its execution until receiving another message is a deterministic function of the contents of the message and the state of the process when the message was received. This does *not* imply that the *order* in which different messages are received is deterministic. In a *nondeterministic* system, no assumption of deterministic process execution is made. Nondeterministic execution can arise, for example, through asynchronous scheduling of multiple threads accessing shared memory.

The execution of each process is divided into a sequence of *state intervals*, which are each identified by a unique *state interval index*. In a deterministic system, a process begins a new state interval each time it receives a new message, and all execution of the process within each state interval is assumed to be deterministic. The state interval index of a process is simply a count of messages received by the process. Each time the process receives a new message, it increments its state interval index, and the new value becomes the index of the state interval started by the receipt of that message. In a nondeterministic system, a process begins a new state interval each time it *sends or receives* a message. The state interval index of a process is a count of these events within the process, and the new value of this counter becomes the index of the state interval started by that send or receive. A process state interval is called *stable* if and only if some state of the process within that interval can be recreated from information on stable storage after a failure.

All messages sent by a process are tagged with the index of the sender's current state interval. When a process receives a message, it then depends on the process state interval from which the message was sent. Each process i records its current dependencies in 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. When a process receives a message, it sets its own dependency vector entry for the sending process to the maximum of its current value and the state interval index tagging the message. Each entry j in process i's dependency vector records the maximum index of any state interval of process j on which process i currently depends. Entry i in process i's own dependency vector records the index of process i's current state interval. If process i has no dependency on any state interval of some process j, then entry j is set to $\bot$, which is less than all possible state interval indices.

A *system state* is a collection of process states, one for each process in the system, and is represented by an $n \times n$ *dependency matrix*

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

where each row i, $\delta_{i\,j}$, $1 \leq j \leq n$, contains the dependency vector for the state of process i included in this system state. Each diagonal element $\delta_{i\,i}$, $1 \leq i \leq n$, of the dependency matrix gives the index of the current state interval of process i. A system state is said to have *occurred* if all component process states have each individually occurred during the preceding execution of the system.

A system state is called *consistent* if and only if no process state records a message as having been received that has not yet been sent in the state of the sender, and that cannot be sent through deterministic execution of the sender from this state. In a nondeterministic system, the future execution of a process is not (assumed to be) deterministic, but in a deterministic system, any messages sent before the end of a process's current state interval can be sent through the deterministic execution of that process. In terms of its dependency matrix, a system state is consistent if and only if no entry in any column is larger than the diagonal entry in that column, indicating that no process depends on a state interval of another process after that process's own current state interval. That is, if $\mathbf{D} = [\delta_{*\,*}]$ represents some system state, then this system state is *consistent* if and only if

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

A system state is called *recoverable* if and only if it is *consistent* and each component process state interval is *stable*.

The system states that have occurred during any single execution of the system may be ordered such that some system state **A** precedes another system state **B**, denoted $\mathbf{A} \prec \mathbf{B}$, if and only if **A** *must* have occurred first during this execution. This partial order can be expressed in terms of the state interval index of each process shown in the dependency matrices representing the system states. That is, if $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ represent two system states that have occurred during the same execution, then

$$\mathbf{A} \preceq \mathbf{B} \iff \forall\, i\, \left[\alpha_{ii} \le \beta_{ii}\right] ,$$

and

$$\mathbf{A} \prec \mathbf{B} \iff (\mathbf{A} \preceq \mathbf{B}) \wedge (\mathbf{A} \neq \mathbf{B}) .$$

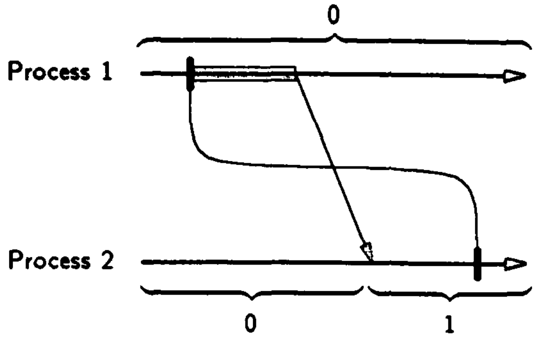
The set of system states that have occurred during any single execution of a system, ordered by this partial order relation, forms a lattice, called the *system history lattice*, and the sets of *consistent* and *recoverable* system states that have occurred during this execution form sublattices of the system history lattice. The proof of this is shown in our previous work with this model [Johnson88], and is omitted here for brevity. Since the set of recoverable system states forms a lattice, there must always be a *unique* maximum recoverable system state, which is simply the least upper bound of all recoverable system states that have occurred. The information recorded on stable storage, making process state intervals stable, must be saved until it is guaranteed not to be needed for any possible future failure recovery in the system [Strom85, Johnson88], and thus the current maximum recoverable system state never decreases.

The distinction between deterministic and nondeterministic systems is captured in the respective definitions of process state intervals, and does not affect the rest of the model. Process state intervals are defined such that all individual states of a process within any single state interval are equivalent in terms of the rest of the model, from the point of view of all other processes in the system. A process can observe the state of another process only when it receives a message sent from that state. In a deterministic system, all individual states of a process within a single state interval result from the deterministic execution of the process within that interval. In a nondeterministic system, only the initial state of a process in any state interval can be observed by other processes.

For example, Figure 1 shows a period of execution in a deterministic system of two processes, and Figure 2 shows the same execution in a nondeterministic system. The horizontal lines represent the execution of each process, and each arrow between processes represents a message sent from one process to the other. The state interval indices of each process are indicated along the lines representing their execution. The state of each process has been checkpointed at each time marked with a vertical bar, making the state intervals in which they are contained stable. Consider the system states formed by taking these stable state intervals, as indicated by the curve connecting the corresponding checkpoints, with the dependency matrices shown. The system state indicated in Figure 1 is consistent because process 1 can deterministically execute from its checkpoint to the state from which it sent the message that is recorded in process 2's checkpoint as having been received. In Figure 2, however, the indicated system state is not consistent, since process 1 cannot guarantee to reach this same state, and thus cannot guarantee to send the same message. This is shown by comparing the circled values in the dependency matrix for this system state.

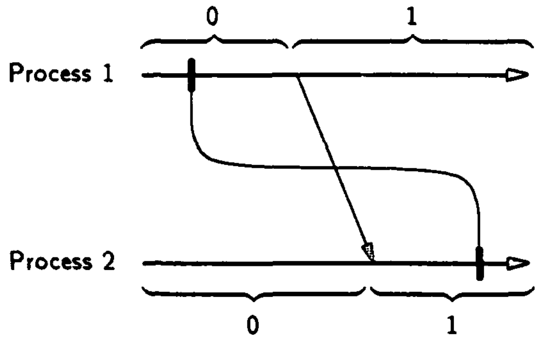
3 Finding the Maximum Recoverable System State

In order to reduce the amount of reexecution of the system necessary to recover after a failure, it is important to recover the system to the maximum possible recoverable system state. Determining the current maximum recoverable system state is also important during failure-free execution, in order to allow output to the



$$\begin{bmatrix} 0 & \perp \\ 0 & 1 \end{bmatrix}$$

Figure 1 An example system state in a deterministic system



$$\begin{bmatrix} \textcircled{0} & \perp \\ \textcircled{1} & 1 \end{bmatrix}$$

Figure 2 An example system state in a nondeterministic system

"outside world" (such as writing information on the user's display terminal) to be committed, and to be able to remove old recovery information from stable storage that is no longer needed [Strom85, Johnson88].

The algorithm presented here guarantees to find the maximum possible recoverable system state at any time. The algorithm requires no additional communication in the system, and requires little storage for execution. We assume that a shared stable storage server in the distributed system is used to record the recovery information during failure-free execution, and the algorithm is designed to execute on that server. The algorithm is restartable if the stable storage server should fail, since all information used by the algorithm has previously been recorded on stable storage. The algorithm can also be easily distributed among a group of stable storage servers used in the system. If additional processes fail while the algorithm is executing, the algorithm is simply restarted. No assumptions are made about the order in which process state intervals become stable, and some state intervals may never become stable. In particular, the fact that some process state interval is stable does *not* imply that all previous state intervals of the same process are also stable. The algorithm can be executed at any time, and considers all process state intervals that are currently stable.

3.1 The Algorithm

The input to the algorithm consists of the dependency vectors for each stable process state interval. Its output consists of the dependency matrix representing the maximum recoverable system state that exists using these stable state intervals. The diagonal elements of this matrix show the index of the state interval of each process contained in this system state. The algorithm finds the maximum recoverable system state "from scratch" each time it is invoked. No internal state is saved between executions of the algorithm.

Conceptually, the algorithm begins its search at the maximum system state that has occurred in which all process state intervals are stable. It then searches backward to lower points in the system history lattice, considering only stable process state intervals, until a consistent system state is found. This system state must then be the maximum recoverable system state. In particular, the following steps are performed by the algorithm:

1. Make a new dependency matrix $\mathbf{D} = [\delta_{*\,*}]$, where each row i is set to the dependency vector for the maximum stable state interval of process i.
2. Loop on step 2 while $\mathbf{D}$ is not consistent. That is, loop while there exists some i and j for which $\delta_{j\,i} > \delta_{i\,i}$, showing that state interval $\delta_{j\,j}$ of process j depends on state interval $\delta_{j\,i}$ of process i, which is greater than process i's current state interval $\delta_{i\,i}$ in $\mathbf{D}$.
    (a) Find the maximum state interval index α less than $\delta_{j\,j}$ of any *stable* state interval of process j such that component i of the dependency vector for this state interval α of process j is not greater than $\delta_{i\,i}$.
    (b) Replace row j of $\mathbf{D}$ with the dependency vector for this state interval α of process j.
3. The system state represented by $\mathbf{D}$ is now consistent and is composed entirely of stable process state intervals. It is now the maximum recoverable system state.

This algorithm can be implemented efficiently by the procedure shown in Figure 3. For each stable process state interval α of each process i, the vector DV_i^α represents the dependency vector for that state interval. The dependency matrix is not explicitly represented; rather, a vector $MAXREC$ is used to store the state interval index of each process in this system state, and the dependency matrix is implicitly represented

```
for $i \leftarrow 1$ to $n$ do
    $MAXREC[i] \leftarrow$ maximum state interval index $\alpha$ of process $i$ such that stable$(i, \alpha)$;
$CHECK \leftarrow \{ i \mid 1 \leq i \leq n \}$;
while $CHECK \neq \emptyset$ do
        $NEW \leftarrow \emptyset$;
        for all $i \in CHECK$ do
                for $j \leftarrow 1$ to $n$ do
                        $\sigma \leftarrow MAXREC[j]$;
                        if $DV_j^\sigma[i] > MAXREC[i]$ then
                            $MAXREC[j] \leftarrow$ maximum state interval index $\alpha < \sigma$ of process $j$
                                                        such that (stable$(j, \alpha)$ $\wedge$ $DV_j^\alpha[i] \leq MAXREC[i]$);
                            $NEW \leftarrow NEW \cup \{ j \}$;
        $CHECK \leftarrow NEW$;
return $MAXREC$;
```

Figure 3 Algorithm for finding the maximum recoverable system state

by the dependency vectors for each of the indicated state intervals. The predicate stable(i, α) is true if and only if state interval α of process i is currently stable.

Each iteration of the **while** loop (lines 4 through 12) performs a pass over the dependency matrix, checking the columns identified by the set *CHECK*. The set *NEW* identifies the columns to be checked on the next iteration. Checking each column i verifies that no process j (in state interval $\sigma = MAXREC[j]$) depends on a state interval of process i that is greater than process i's current state interval in the dependency matrix, $MAXREC[i]$. If such a process j is found (line 9), its row in the dependency matrix is implicitly replaced by changing $MAXREC[j]$ to the index of some previous stable state interval α of process j. All dependencies of other processes on process j must then be checked on the next pass over the matrix by the **while** loop, and j is thus added to the set *NEW*.

In a system using a group of stable storage servers, each recording information for the stable process state intervals of a disjoint subset of the processes of the system, the algorithm can be distributed among these servers. In such a system, the rows of the dependency matrix used by the algorithm can be partitioned among the servers such that each row is assigned to the server recording the information (including the dependency vector) for the corresponding process. The main algorithm is executed on any one server, and for each iteration of the **while** loop, the values of *CHECK* and *MAXREC* are sent to each other server. Each server then performs the portion of the **for** loop at lines 7 through 11 to check the entries in the columns identified by *CHECK* in the rows assigned to that server. The results are then returned to the main server for the next iteration of the **while** loop.

3.2 Correctness

In this section, we show the correctness of the algorithm presented in Figure 3. That is, the algorithm completes each execution with each $MAXREC[i]$ containing the state interval index of process i in the current maximum recoverable system state.

The following loop invariant is maintained at the beginning of each iteration of the **for** j loop at line 7:

> If the vector *MAXREC* represents system state $\mathbf{D}$, then no recoverable system state $\mathbf{R}$ currently exists such that $\mathbf{D} \prec \mathbf{R}$.

Before the first iteration of the loop, this invariant must hold, since the system state $\mathbf{D}$ is the maximum system state that currently exists having each component process state interval stable. Assuming that the invariant holds at the beginning of some iteration, execution of the loop body preserves the invariant. If for the current i and j at line 9, $DV_j^\sigma[i] \leq MAXREC[i]$, then *MAXREC* remains unchanged and the invariant is preserved. Otherwise, $DV_j^\sigma[i] > MAXREC[i]$, and the system state represented by $\mathbf{D}$ is thus not consistent. Process j (in state interval $\sigma = MAXREC[j]$) depends on state interval $DV_j^\sigma[i]$ of process i, but process i in $\mathbf{D}$ is only in state interval $MAXREC[i]$. In any recoverable system state that exists, process j must not depend on any state interval of process i greater than $MAXREC[i]$. The loop invariant is thus maintained in this case by choosing the *largest* state interval index $\alpha < \sigma$ of process j such that state interval α of process j is stable and does not depend on any state interval of process i greater than $MAXREC[i]$.

If the **while** loop terminates after line 12, the system state found must be consistent, since no rows of the dependency matrix were replaced during its final iteration. The predicate on the **if** statement (line 9) tests each dependency according to the definition of a consistent system state. This system state must also be recoverable, since each process state interval included in it by the **for** loop at the beginning of the algorithm (line 2) is stable, and only stable process state intervals are used to replace components of it during the execution of the **while** loop.

At each iteration of the **for** loop at line 7, the system state **D** being considered precedes its value from the previous iteration. The search by the algorithm begins at the maximum system state, **D**, composed of only stable process state intervals, and thus no recoverable system state $\mathbf{R}'$ can exist such that $\mathbf{D} \prec \mathbf{R}'$. However, *some* recoverable system state must exist in the system, since recovery information is not removed from stable storage until no longer needed for any possible future failure recovery. Thus, by the loop invariant, the algorithm must terminate. The system state **D** represented by *MAXREC* must be a recoverable system state, and must therefore be the *maximum* recoverable system state that currently exists.

4 Implementation Experience

We have implemented the algorithm shown in Figure 3 in a system using optimistic message logging and checkpointing [Johnson89], running under the V-System [Cheriton83, Cheriton88]. Each process saves messages as they are received in a local buffer in volatile memory. When the buffer fills or after a specified timeout has expired, the buffer is written to stable storage, logging these messages. All other details of message logging, including replaying logged messages during recovery, are handled by a *logging server* running on each machine. Each process is also occasionally checkpointed to stable storage, but there is no coordination between the checkpointing of individual processes or between checkpointing and message logging. In particular, a process may record a new checkpoint without logging all previously received messages, and if the maximum recoverable system state subsequently advances beyond the receipt of those message (using the stable state interval recorded in the checkpoint), those messages need not be logged. The V-System kernel initiates checkpointing once the process has received a given number of messages or has consumed a given amount of processor time since its last checkpoint. The creation of the checkpoint, and the restoration of the process from its checkpoint during recovery, are handled by a *checkpoint server* running on each machine. All logged messages and checkpoints are saved on the shared network file server used by all processes in the system.

The algorithm is implemented by a single *recovery server* running on the shared network file server machine. For each process, the recovery server maintains a linked list describing its stable state intervals, in descending order by state interval index. Since message logging is not coordinated with checkpointing, the stable state intervals of each process can be divided into groups, each beginning with a state interval recorded in a checkpoint, followed by consecutive state intervals for which the message starting each has been logged. These groups are each represented by a single entry in the linked list. The complete dependency vector is stored only for the last state interval in each group, and for each previous entry in a group, only the single difference between that state interval's dependency vector and the vector for the next higher state interval is stored. As the algorithm searches backward for the needed state interval α of process j (at line 10 in Figure 3), a copy of the dependency vector is modified from these differences to efficiently construct the dependency vector for state interval α.

We currently have only limited experience with the implementation of the algorithm. We have used the optimistic message logging and checkpointing system in executing a set of distributed application programs to solve a number of different problems. These applications include programs for solving the n-queens problem, the traveling salesman problem, and Gaussian elimination with partial pivoting. The overhead of message logging and checkpointing during failure-free execution of these application programs ranged from a maximum of under 4 percent to much less than 1 percent. During failure recovery, the running time of the algorithm is negligible relative to the time required to restore the processes from their checkpoints and to replay the logged messages to the recovering processes.

5 Related Work

Strom and Yemini introduced the notion of optimistic message logging and checkpointing [Strom85, Strom88]. Each process in their system maintains a *transitive* dependency vector, recording the transitive closure of the dependencies represented by our dependency vectors. A copy of the sender's dependency vector is included with each message sent, and is then merged with the receiver's vector when the message is received. In our system, only the sender's current state interval index is included with each message. Strom and Yemini also require each process to maintain 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. Each process then determines its own state interval in the maximum recoverable system state by a comparison of the local dependency vector and log vector. Although this allows their algorithm to be completely distributed, the additional communication required in their system can add substantially to the failure-free overhead of the system. Our algorithm also makes use of the stable process state intervals recorded in checkpoints to advance the maximum recoverable system state, whereas their algorithm only advances when messages are logged. If the maximum recoverable system state advances due to a process checkpoint, before previously received messages have been logged, those messages need never be logged.

Sistla and Welch have proposed two alternative recovery algorithms based on optimistic message logging [Sistla89]. 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 previous work [Johnson88]. To find the maximum recoverable system 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 complete maximum recoverable system 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. Again, this additional communication allows their algorithm to be completely distributed, but adds significantly to the failure-free overhead of the system. Their algorithm also does not use the process checkpoints to advance the maximum recoverable system state, as ours does.

The algorithm presented here is similar to our previous algorithm for use with optimistic message logging and checkpointing [Johnson88]. That algorithm *incrementally* determines the maximum recoverable system state from the previous maximum. If no new recoverable system state exists after some new process state interval becomes stable, that state interval is added to a number of "defer" sets to be rechecked later. Although this shortens the search for the maximum recoverable system state, the additional overhead of maintaining and rechecking these defer sets may offset much of this advantage. If the algorithm must be executed frequently (such as to allow output to the "outside world" to be committed), the incremental algorithm may be preferable, but if executions are infrequent, the algorithm presented here may be preferable. The relative simplicity of the algorithm presented here also makes it attractive. We have not yet specifically quantified the circumstances under which each of these two algorithms is preferable.

Bhargava and Lian describe an optimistic rollback recovery method using checkpointing alone, without message logging [Bhargava88]. Processes maintain a *checkpoint number* and an *input information table*, roughly serving the role of our state interval indices and dependency vectors. Each message sent includes the current checkpoint number of the sender. For a process to determine the maximum recoverable system state, it first requests the input information table from each other process. It then uses this information to incrementally update a *local system graph* showing its view of the message communication in the system, and performs a depth-first search on this graph. The input information table records the checkpoint numbers of *all* messages received by the process since the last known maximum recoverable system state, whereas our dependency vectors store *only* the maximum state interval index of any message received from each process. Also, the data structures used internally by our algorithm (*MAXREC* and the dependency vectors) require less storage than their local system graph, and do not require an explicit construction phase in the algorithm. Their algorithm, though, can be used concurrently with new failures occurring in the system, whereas our algorithm must be restarted if a new failure occurs while it is executing.

Although the model and algorithm presented in this paper can be used in *pessimistic* rollback recovery methods, their full generality is not required, since the maximum recoverable system state is readily determinable from the synchronization imposed on the system. For example, with pessimistic message logging and checkpointing methods [Powell83, Johnson87, Borg89], the maximum recoverable system state is simply composed of the most recent state interval of each process for which all previously received messages have been logged. With pessimistic checkpointing methods [Koo87, Chandy85], a complete *global checkpoint* of the system is maintained by the checkpointing protocol, such that the checkpoints of all process in this global checkpoint form a consistent system state. The most recent global checkpoint is thus the maximum recoverable system state. On the other hand, the disadvantage of pessimistic methods over optimistic methods is the overhead of the additional synchronization required to achieve this simplicity of determining the maximum recoverable system state.

6 Conclusion

This paper has presented a simple algorithm that guarantees to find the *maximum* possible recoverable system state in any system using optimistic rollback recovery. The algorithm can be used in systems in which all process execution between received messages is assumed to be deterministic, as well as in systems making no such assumption. For example, the algorithm can be used in deterministic systems using a fault-tolerance method based on optimistic message logging and checkpointing [Strom85, Strom88, Johnson88, Sistla89, Johnson89], and in nondeterministic systems using optimistic checkpointing alone [Bhargava88]. The distinction between deterministic and nondeterministic systems is captured in the system model by the definition of process *state intervals*, such that all individual states of a process within any single state interval are equivalent in terms of the rest of the model and the algorithm, from the point of view of all other processes in the system.

We have implemented the algorithm in a system using optimistic message logging and checkpointing, running under the V-System [Cheriton83, Cheriton88]. A shared network file server is used to record all information for stable storage, and the algorithm is executed on that server. The algorithm is restartable if the file server should fail, since all information used by the algorithm has previously been recorded on stable storage. Likewise, if additional processes fail while the algorithm is executing, the algorithm is simply restarted. In our experience with this implementation, the running time of the algorithm is negligible relative to the time required to restore the processes from their checkpoints during recovery and to replay the logged messages to the recovering processes.

References

[Bhargava88] Bharat Bhargava and Shy-Renn Lian. Independent checkpointing and concurrent rollback for recovery—An optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 3–12. IEEE Computer Society, October 1988.

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Chandy85] 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.

[Cheriton83] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[Cheriton88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Johnson87] 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.

[Johnson88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[Johnson89] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. Ph.D. thesis, Rice University, Houston, Texas, December 1989.

[Koo87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Powell83] 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.

[Sistla89] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 1989.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Strom88] Robert E. Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49. IEEE Computer Society, June 1988.