A Survey of Rollback-Recovery Protocols in Message-Passing Systems

E. N. (MOOTAZ) ELNOZAHY

IBM Research

LORENZO ALVISI

The University of Texas at Austin

YI-MIN WANG

Microsoft Research

AND

DAVID B. JOHNSON

Rice University

This survey covers rollback-recovery techniques that do not require special language constructs. In the first part of the survey we classify rollback-recovery protocols into *checkpoint-based* and *log-based. Checkpoint-based* protocols rely solely on checkpointing for system state restoration. Checkpointing can be coordinated, uncoordinated, or communication-induced. *Log-based* protocols combine checkpointing with logging of nondeterministic events, encoded in tuples called *determinants*. Depending on how determinants are logged, log-based protocols can be pessimistic, optimistic, or causal. Throughout the survey, we highlight the research issues that are at the core of rollback-recovery and present the solutions that currently address them. We also compare the performance of different rollback-recovery protocols with respect to a series of desirable properties and discuss the issues that arise in the practical implementations of these protocols.

Categories and Subject Descriptors: D.4.5 **[Operating Systems]**: Reliability—*Checkpoint/restart*; *fault-tolerance*; D.4.7 **[Operating Systems]**: Organization and Design—*Distributed systems*; D.2.8 **[Software]**: Metrics—*Performance measures*;

General Terms: Design, Reliability, Performance

Additional Key Words and Phrases: message logging, rollback-recovery

---

Mootaz Elnozahy started this work while at Carnegie Mellon University, where he was supported in part by the National Science Foundation through a Research Initiation Award under contract CCR 9410116 and a CAREER Award under contract CCR 9502933. Lorenzo Alvisi was supported in part by an NSF CAREER award (CCR-9734185), an Alfred P. Sloan Fellowship, an IBM Faculty Partnership award, DARPA/SPAWAR grant N66001-98-8911, and a grant of the Texas Advanced Research Program.

Authors' addresses: E. N. (Mootaz) Elnozahy, IBM Austin Research Lab., M/S 904-6C-020, 11501 Burnet Rd., Austin, TX 78578; email: mootaz@us.ibm.com; Lorenzo Alvisi, Department of Computer Sciences, Taylor Hall 2.124 The University of Texas at Austin, Austin, TX 78712-1188; email: lorenzo@cs.utexas.edu; Yi-Min Wang, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052; email: ymwang@microsoft.com; David B. Johnson, Rice University, Department of Computer Science, 6100 Main St., MS 132, Houston, TX 77005-1892; email: dbj@cs.rice.edu.

1. INTRODUCTION

Distributed systems today are ubiquitous and enable many applications, including client-server systems, transaction processing, World Wide Web, and scientific computing, among many others. The vast computing potential of these systems is often hampered by their susceptibility to failures. Therefore, many techniques have been developed to add reliability and high availability to distributed systems. These techniques include transactions, group communication, and rollback-recovery, and have different tradeoffs and focuses. For example, transactions focus on data-oriented applications, while group communication offers an abstraction of an ideal communication system that simplifies the development of reliable applications. This survey covers transparent rollback-recovery, which focuses on long-running applications such as scientific computing and telecommunication applications [Huang and Kintala 1993; Plank 1993].

Rollback-recovery treats a distributed system as a collection of application processes that communicate through a network. The processes have access to a *stable storage* device that survives all tolerated failures. Processes achieve fault tolerance by using this device to save recovery information periodically during failure-free execution. Upon a failure, a failed process uses the saved information to restart the computation from an intermediate state, thereby reducing the amount of lost computation. The recovery information includes, at a minimum, the states of the participating processes, called *checkpoints*. Other recovery protocols may require additional information, such as logs of the interactions with input and output devices, events that occur to each process, and messages exchanged among the processes.

Rollback-recovery has many flavors. For example, a system may rely on the application to decide when and what to save on stable storage. Or, it may provide the application programmer with linguistic constructs to structure the application [Randell 1975]. We focus in this survey on *transparent* techniques, which do not require any intervention on the part of the application or the programmer. The system automatically takes checkpoints according to some specified policy, and recovers automatically from failures if they occur. This approach has the advantages of relieving the application programmers from the complex and error-prone chores of implementing fault tolerance and of offering fault tolerance to existing applications written without consideration to reliability concerns.

Rollback-recovery has been studied in various forms and in connection with many fields of research. Thus, it is perhaps impossible to provide an extensive coverage of all the issues related to rollback-recovery within the scope of one article. This survey concentrates on the definitions, fundamental concepts, and implementation issues of rollback-recovery protocols in distributed systems. The coverage excludes the use of rollback-recovery in many related fields such as hardware-level instruction retry, distributed shared memory [Morin and Puaut 1997], real-time systems, and debugging [Mellor-Crummey and LeBlanc 1989]. The coverage also excludes the issues of using rollback-recovery when failures could include Byzantine modes or are not restricted to the fail-stop model [Schlichting and Schneider 1983]. Also excluded are rollback-recovery techniques that rely on special language constructs such as recovery blocks [Randell 1975] and transactions. Finally, the section on implementation exposes many relevant issues related to implementing checkpointing on uniprocessors, although the coverage is by no means an exhaustive one because of the large number of issues involved.

Message-passing systems complicate rollback-recovery because messages induce inter-process dependencies during failure-free operation. Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called *rollback propagation*. To see why rollback

propagation occurs, consider the situation where a sender of a message m rolls back to a state that precedes the sending of m. The receiver of m must also roll back to a state that precedes m's receipt; otherwise, the states of the two processes would be *inconsistent* because they would show that message m was received without being sent, which is impossible in any correct failure-free execution. Under some scenarios, rollback propagation may extend back to the initial state of the computation, losing all the work performed before a failure. This situation is known as the *domino effect* [Randell 1975].

The domino effect may occur if each process takes its checkpoints independently—an approach known as *independent* or *uncoordinated checkpointing*. It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it. One such technique is to perform *coordinated checkpointing* in which processes coordinate their checkpoints in order to save a system-wide consistent state [Chandy and Lamport 1985]. This consistent set of checkpoints can then be used to bound rollback propagation. Alternatively, *communication-induced checkpointing* forces each process to take checkpoints based on information piggybacked on the application messages received from other processes [Russell 1980]. Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

The approaches discussed so far implement *checkpoint-based* rollback-recovery, which relies only on checkpoints to achieve fault-tolerance. In contrast, *log-based* rollback-recovery combines checkpointing with logging of nondeterministic events.[1] Log-based rollback-recovery relies on the *piecewise deterministic* (*PWD*) assumption [Strom and Yemini 1985], which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's *determinant* [Alvisi 1996; Alvisi and Marzullo 1998]. By logging and replaying the nondeterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed. Log-based rollback-recovery in general enables a system to recover beyond the most recent set of consistent checkpoints. It is therefore particularly attractive for applications that frequently interact with the *outside world*, which consists of all input and output devices that cannot roll back. Log-based rollback-recovery has three flavors, depending on how the determinants are logged to stable storage. In *pessimistic logging*, the application has to block waiting for the determinant of each nondeterministic event to be stored on stable storage before the effects of that event can be seen by other processes or the outside world. Pessimistic logging simplifies recovery but hurts failure-free performance. In *optimistic logging*, the application does not block, and determinants are spooled to stable storage asynchronously. Optimistic logging reduces the failure-free overhead, but complicates recovery. Finally, in *causal logging*, low failure-free overhead and simpler recovery are combined by striking a balance between optimistic and pessimistic logging. The three flavors also differ in their requirements for garbage collection and their interactions with the outside world, as will be explained later.

The outline of the rest of the survey is as follows:

- Section 2: System model, terminology and generic issues in rollback-recovery.
- Section 3: Checkpoint-based rollback-recovery protocols.
- Section 4: Log-based rollback-recovery protocols.
- Section 5: Implementation issues.
- Section 6: Conclusions.

[1] Earlier papers in this area have assumed a model in which the occurrence of a nondeterministic event is modeled as a message receipt. In this model, nondeterministic event logging reduces to *message logging*. In this paper, we use the terms event logging and message logging interchangeably.

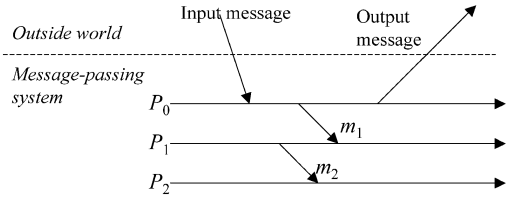

Fig. 1. An example of a message-passing system with three processes.

2. BACKGROUND AND DEFINITIONS

2.1. System Model

A message-passing system consists of a fixed number of processes that communicate only through messages. Throughout this survey, we use N to denote the total number of processes in a system. Processes cooperate to execute a distributed application program and interact with the outside world by receiving and sending input and output messages, respectively. Figure 1 shows a sample system consisting of three processes, where horizontal lines extending toward the right-hand side represent the execution of each process, and arrows between processes represent messages.

Rollback-recovery protocols generally assume that the communication network is immune to partitioning but differ in the assumptions they make about network reliability. Some protocols assume that the communication subsystem delivers messages reliably, in first-in-first-out (FIFO) order [Chandy and Lamport 1985], while other protocols assume that the communication subsystem can lose, duplicate, or reorder messages [Johnson 1989]. The choice between these two assumptions usually affects the complexity of recovery and its implementation in different ways. Generally, assuming a reliable network simplifies the design of the recovery protocol but introduces implementation complexities that will be described in Sections 2.3, 2.4 and 5.4.2.

A process execution is a sequence of *state intervals*, each started by a nondeterministic event. Execution during each state interval is deterministic, such that if a process starts from the same state and is subjected to the same nondeterministic events at the same locations within the execution, it will always yield the same output. A concept related to the state interval is the piecewise deterministic assumption (PWD). This assumption states that the system can detect and capture sufficient information about the nondeterministic events that initiate the state intervals.

A process may fail, in which case it loses its volatile state and stops execution according to the fail-stop model [Schlichting and Schneider 1983]. Processes have access to a stable storage device that survives failures, such that state information saved on this device during failure-free execution can be used for recovery. The number of tolerated process failures may vary from 1 to N, and the recovery protocol needs to be designed accordingly. Furthermore, some protocols may not tolerate failures that occur during recovery.

A generic correctness condition for rollback-recovery can be defined as follows: "A system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure" [Strom and Yemini 1985]. Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world. A description of the notion of consistency and the interactions with the outside world follows.

2.2. Consistent System States

A global state of a message-passing system is a collection of the individual states of all participating processes and of the states of the communication channels. Intuitively, a consistent global state is one that may occur during a failure-free, correct execution of a distributed computation. More precisely, a *consistent system state* is one in which, if the state of a process reflects a message receipt, then the state of the corresponding sender reflects sending that message [Chandy and Lamport 1985]. For example, Figure 2 shows two examples of global states—a consistent state in Figure 2(a), and an inconsistent

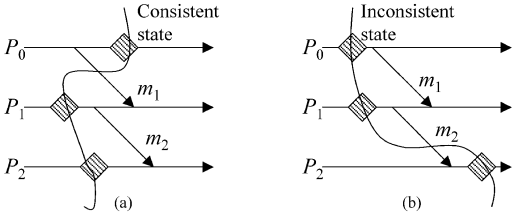

Fig. 2. An example of a consistent and inconsistent state.

state in Figure 2(b). Note that the consistent state in Figure 2(a) shows message m_1 to have been sent but not yet received. This state is consistent, because it represents a situation in which the message has left the sender and is still traveling across the network. On the other hand, the state in Figure 2(b) is inconsistent because process P_2 is shown to have received m_2 but the state of process P_1 does not reflect sending it. Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures. For example, the situation shown in part (b) of Figure 2 may occur if process P_1 fails after sending message m_2 to P_2 and then restarts at the state shown in the figure.

A fundamental goal of any rollback-recovery protocol is to bring the system into a consistent state when inconsistencies occur because of a failure. The reconstructed consistent state is not necessarily one that has occurred before the failure. It is sufficient that the reconstructed state be one that *could* have occurred before the failure in a failure-free, correct execution, provided that it be consistent with the interactions that the system had with the outside world. We describe these interactions next.

2.3. Interactions with the Outside World

A message-passing system often interacts with the outside world to receive input data or show the outcome of a computation. If a failure occurs, the outside world cannot be relied on to roll back [Pausch 1988]. For example, a printer cannot roll back the effects of printing a character, and an automatic teller machine cannot recover the money that it dispensed to a customer. To simplify the presentation of how rollback-recovery protocols interact with the outside world, we model the latter as a *special* process that interacts with the rest of the system through message passing. This special process cannot fail, and it cannot maintain state or participate in the recovery protocol. Furthermore, since this special process models irreversible effects in the outside world, it cannot roll back. We call this special process the "outside world process" (*OWP*).

It is necessary that the outside world perceive a consistent behavior of the system despite failures. Thus, before sending a message (output) to OWP, the system must ensure that the state from which the message is sent will be recovered despite any future failure. This is commonly called the *output commit* problem [Strom and Yemini 1985]. Similarly, input messages that a system receives from the outside world may not be reproducible during recovery, because it may not be possible for OWP to regenerate them. Thus, recovery protocols must arrange to save these input messages so that they can be retrieved when needed for execution replay after a failure. A common approach is to save each input message on stable storage before allowing the application program to process it.

Rollback-recovery protocols, therefore, must provide special treatment for interactions with the outside world. There are two metrics that express the impact of this special treatment, namely the latency of input/output and the resulting slowdown of system's execution during input/output. The first metric represents the time it takes for an output message to be released to OWP after it has been issued by the system, or the time it takes a process to consume an input message after it has been sent from OWP. The second metric represents the overhead that the system incurs to ensure that its state will remain consistent with the messages exchanged with the OWP despite future failures.

2.4. In-Transit Messages

In Figure 2(a), the global state shows that message m_1 has been sent but not

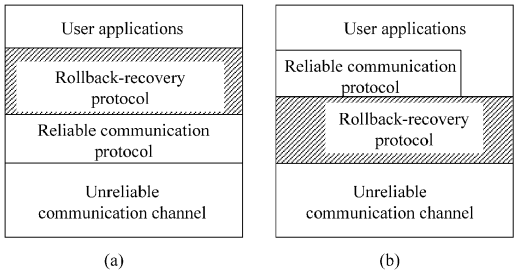

Fig. 3. Implementation of rollback-recovery (a) on top of a reliable communication protocol; (b) directly on top of unreliable communication channels.

yet received. We call such a message an *in-transit* message. When in-transit messages are part of a global system state, they do not cause any inconsistency. However, depending on whether the system model assumes reliable communication channels, rollback-recovery protocols may have to guarantee the delivery of in-transit messages when failures occur. For example, the rollback-recovery protocol in Figure 3(a) assumes reliable communications, and therefore it must be implemented on top of a reliable communication protocol layer. In contrast, the rollback-recovery protocol in Figure 3(b) does not assume reliable communications.

Reliable communication protocols ensure the reliability of message delivery during failure-free executions. They cannot, however, ensure by themselves, the reliability of message delivery in the presence of process failures. For instance, if an in-transit message is lost because the intended receiver has failed, conventional communication protocols will generate a timeout and inform the sender that the message cannot be delivered. In a rollback-recovery system, however, the receiver will eventually recover. Therefore, the system must mask the timeout from the application program at the sender process and must make in-transit messages available to the intended receiver process after it recovers, in order to ensure a consistent view of the reliable system. On the other hand, if a system model assumes unreliable communication channels, as in Figure 3(b), then the recovery protocol need not handle in-transit messages in any special way. Indeed, in-transit messages lost because of process failures cannot be distinguished from those lost because of communication failures in an unreliable communication channel. Therefore, the loss of in-transit messages due to either communication or process failure is an event that can occur in any failure-free, correct execution of the system.

2.5. Logging Protocols

Log-based rollback-recovery uses checkpointing and logging to enable processes to replay their execution after a failure beyond the most recent checkpoint. This is useful when interactions with the outside world are frequent, since it enables a process to repeat its execution and be consistent with messages sent to OWP without having to take expensive checkpoints before sending such messages. Additionally, log-based recovery generally is not susceptible to the domino effect, thereby allowing processes to use uncoordinated checkpointing if desired.

Log-based recovery relies on the *piecewise deterministic* (*PWD*) assumption [Strom and Yemini 1985]. Under this assumption, the rollback-recovery protocol can identify all the nondeterministic events executed by each process, and for each such event, logs a *determinant* that contains all information necessary to replay the event should it be necessary during recovery. If the PWD assumption holds, log-based rollback-recovery protocols can recover a failed process and replay its execution as it occurred before the failure.

Examples of nondeterministic events include receiving messages, receiving input from the outside world, or undergoing an internal state transfer within a process based on some nondeterministic action such as the receipt of an interrupt. Rollback-recovery implementations differ in the range of actual nondeterministic events that are covered under this model. For instance, a particular implementation may only cover message receipts from other processes under the PWD assumption. Such an implementation cannot

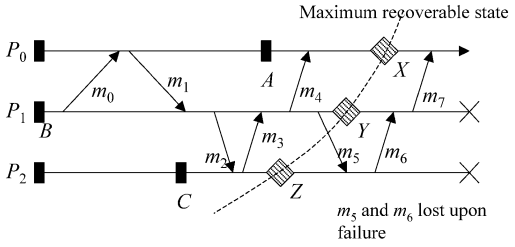

Fig. 4. Message logging for deterministic replay.

replay an execution that is subjected to other forms of nondeterministic events such as asynchronous interrupts. The range of events covered under the PWD assumption is an implementation issue and is covered in Section 5.7.

A state interval is *recoverable* if there is sufficient information to replay the execution up to that state interval despite any future failures in the system. Also, a state interval is *stable* if the determinant of the nondeterministic event that started it is logged on stable storage [Johnson and Zwaenepoel 1990]. A recoverable state interval is always stable, but the opposite is not always true [Johnson 1989].

Figure 4 shows an execution in which the only nondeterministic events are message deliveries. Suppose that processes P_1 and P_2 fail before logging the determinants corresponding to the deliveries of m_6 and m_5, respectively, while all other determinants survive the failure. Message m_7 becomes an *orphan message* because process P_2 cannot guarantee the regeneration of the same m_6 during recovery, and P_1 cannot guarantee the regeneration of the same m_7 without the original m_6. As a result, the surviving process P_0 becomes an *orphan process* and is forced to roll back as well. States X, Y and Z form the *maximum recoverable state* [Johnson 1989], that is, the most recent recoverable consistent system state. Processes P_0 and P_2 roll back to checkpoints A and C, respectively, and replay the deliveries of messages m_4 and m_2, respectively, to reach states X and Z. Process P_1 rolls back to checkpoint B and replays the deliveries of m_1 and m_3 in their original order to reach state Y.

During recovery, log-based rollback-recovery protocols force the execution of the system to be identical to the one that occurred before the failure, up to the maximum recoverable state. Therefore, the system always recovers to a state that is consistent with the input and output interactions that occurred up to the maximum recoverable state.

2.6. Stable Storage

Rollback-recovery uses stable storage to save checkpoints, event logs, and other recovery-related information. Stable storage in rollback-recovery is only an abstraction, although it is often confused with the disk storage used to implement it. Stable storage must ensure that the recovery data persist through the tolerated failures and their corresponding recoveries. This requirement can lead to different implementation styles of stable storage:

- In a system that tolerates only a single failure, stable storage may consist of the volatile memory of another process [Borg et al. 1989; Johnson and Zwaenepoel 1987].
- In a system that wishes to tolerate an arbitrary number of *transient* failures, stable storage may consist of a local disk in each host.
- In a system that tolerates non-transient failures, stable storage must consist of a persistent medium outside the host on which a process is running. A replicated file system is a possible implementation in such systems [Lampson and Sturgis 1979].

2.7. Garbage Collection

Checkpoints and event logs consume storage resources. As the application progresses and more recovery information is collected, a subset of the stored information may become useless for recovery. Garbage collection is the deletion of such useless recovery information. A common approach to garbage collection is to identify the most recent consistent set of checkpoints, which is called the *recovery*

line [Randell 1975], and discard all information relating to events that occurred before that line. For example, processes that coordinate their checkpoints to form consistent states will always restart from the most recent checkpoint of each process, and so all previous checkpoints can be discarded. While it has received little attention in the literature, garbage collection is an important pragmatic issue in rollback-recovery protocols, because running a special algorithm to discard useless information incurs overhead. Furthermore, recovery-protocols differ in the amount and nature of the recovery information they need to store on stable storage, and therefore differ in the complexity and invocation frequency of their garbage collection algorithms.

3. CHECKPOINT-BASED ROLLBACK RECOVERY

Upon a failure, checkpoint-based rollback-recovery restores the system state to the most recent consistent set of checkpoints, that is the recovery line [Randell 1975]. It does not rely on the PWD assumption, and so does not need to detect, log, or replay nondeterministic events. Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback-recovery. But checkpoint-based rollback-recovery does not guarantee that pre-failure execution can be deterministically regenerated after a rollback. Therefore, checkpoint-based rollback-recovery is ill suited for applications that require frequent interactions with the outside world, since such interactions require that the observable behavior of the system through failures and recoveries be the same as during a failure-free execution. Checkpoint-based rollback-recovery techniques can be classified into three categories: *uncoordinated checkpointing*, *coordinated checkpointing*, and *communication-induced checkpointing*. We examine each category in detail.

3.1. Uncoordinated Checkpointing

3.1.1. Overview. Uncoordinated checkpointing allows each process the maximum autonomy in deciding when to take checkpoints. The main advantage of this autonomy is that each process may take a checkpoint when it is most convenient. For example, a process may reduce the overhead by taking checkpoints when the amount of state information to be saved is small [Wang 1993]. But there are several disadvantages. First, there is the possibility of the domino effect, which may cause the loss of a large amount of useful work, possibly all the way back to the beginning of the computation. Second, a process may take a *useless* checkpoint that will never be part of a global consistent state. Useless checkpoints are undesirable because they incur overhead and do not contribute to advancing the recovery line. Third, uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer useful. Fourth, it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

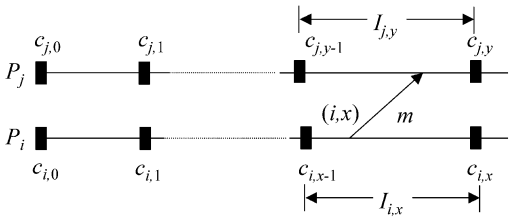

Fig. 5. Checkpoint index and checkpoint interval.

In order to determine a consistent global checkpoint during recovery, the processes record the dependencies among their checkpoints during failure-free operation using the following technique [Bhargava and Lian 1988]. Let $c_{i,x}$ be the x the checkpoint of process P_i. We call x the *checkpoint index*. Let $I_{i,x}$ denote the *checkpoint interval* or simply *interval* between checkpoints $c_{i,x-1}$ and $c_{i,x}$. As illustrated in Figure 5, if process P_i at interval $I_{i,x}$ sends a message m to P_j, it will piggyback the pair (i, x) on m. When P_j receives m during interval $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto

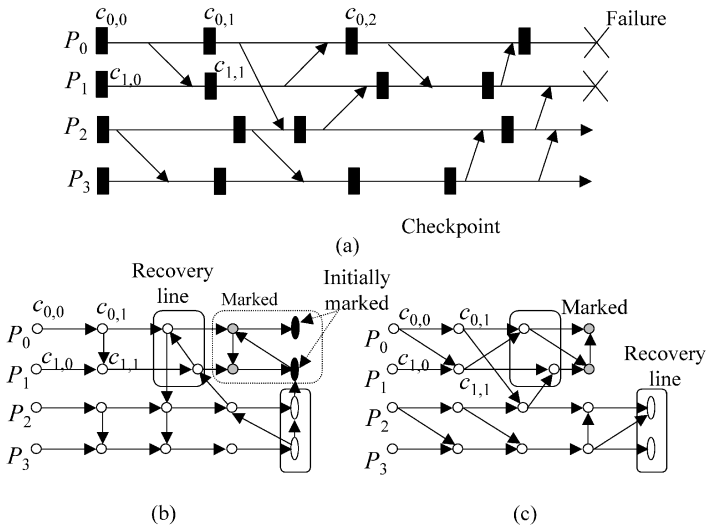

Fig. 6. (a) Example execution; (b) rollback-dependency graph; (c) checkpoint graph.

stable storage when P_j takes checkpoint $c_{j,y}$.

If a failure occurs, the recovering process initiates rollback by broadcasting a *dependency request* message to collect all the dependency information maintained by each process. When a process receives this message, it stops its execution and replies with the dependency information saved on stable storage as well as with the dependency information, if any, that is associated with its current state. The initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line. Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise it rolls back to an earlier checkpoint as indicated by the recovery line.

3.1.2. Dependency Graphs and Recovery Line Calculation. There are two approaches proposed in the literature to determine the recovery line in checkpoint-based recovery. The first approach is based on a *rollback-dependency graph* [Bhargava and Lian 1988] in which each node represents a checkpoint and a directed edge is drawn from $c_{i,x}$ to $c_{j,y}$ if either:

(1) $i \neq j$, and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$, or

(2) $i = j$ and $y = x + 1$.

The name "rollback-dependency graph" comes from the observation that if there is an edge from $c_{i,x}$ to $c_{j,y}$ and a failure forces $I_{i,x}$ to be rolled back, then $I_{j,y}$ must also be rolled back.

Figure 6(b) shows the rollback dependency graph for the execution in Figure 6(a). The algorithm used to compute the recovery line first marks the graph nodes corresponding to the states of processes P_0 and P_1 at the failure point (shown in figure in dark ellipses). It then uses reachability analysis [Bhargava and Lian 1988] to mark all reachable nodes from any of the initially marked nodes. The union of the *last* unmarked nodes over the entire system forms the recovery line, as shown in Figure 6(b).

The second approach is based on the *checkpoint graph* [Wang 1993]. Checkpoint graphs are similar to rollback-dependency graphs except that, when a message is sent from $I_{i,x}$ and received in $I_{j,y}$, a directed edge is drawn from $c_{i,x-1}$ to $c_{j,y}$ (instead of $c_{i,x}$ to $c_{j,y}$), as shown in Figure 6(c). The recovery line can be calculated by first removing both the nodes

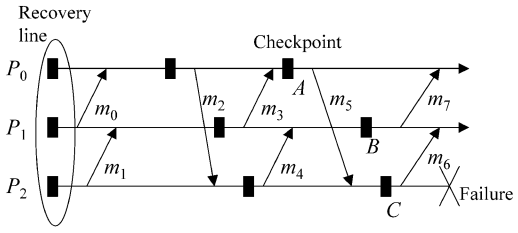

Fig. 7. Rollback propagation, recovery line and the domino effect.

corresponding to the states of the failed processes at the point of failures and the edges incident on them, and then applying the *rollback propagation algorithm* [Wang 1993] on the checkpoint graph. Both the rollback-dependency graph and the checkpoint graph approaches are equivalent, in that they always produce the same recovery line (as indeed they do in the example). These methods form the basis for performing garbage collection in independent checkpointing, by determining the most advanced recovery line and removing the checkpoints that precede it [Wang 1993]. Additionally, some checkpoints taken independently by a process may never be part of a consistent state and therefore will be useless for recovery purposes. These checkpoints also can be removed using the algorithm described by Wang [1993]. Finally, it can be shown under independent checkpointing that the maximum number of useful checkpoints that must be kept on stable storage cannot exceed $(N\ (N+1)/2)$ [Wang et al. 1995a].

3.1.3. The Domino Effect. While simple to implement, uncoordinated checkpointing can lead to the *domino effect* [Randell 1975]. For example, Figure 7 shows an execution in which processes take their checkpoints—represented by black bars—without coordinating with each other. Each process starts its execution with an initial checkpoint. Suppose process P_2 fails and rolls back to checkpoint C. The rollback "invalidates" the sending of message m_6, and so P_1 must roll back to checkpoint B to "invalidate" the receipt of that message. Thus, the invalidation of message m_6 propagates the rollback of process P_2 to process P_1, which in turn "invalidates" message m_7 and forces P_0 to roll back as well.

This cascaded rollback may continue and eventually may lead to the domino effect, which causes the system to roll back to the beginning of the computation, in spite of all the saved checkpoints. In the example shown in Figure 7, cascading rollbacks due to the single failure of process P_2 forces the system to restart from the initial set of checkpoints, effectively causing the loss of all the work done by all processes.

3.2. Coordinated Checkpointing

3.2.1. Overview. Coordinated checkpointing requires processes to orchestrate their checkpoints in order to form a consistent global state. Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint. Also, coordinated checkpointing requires each process to maintain only one permanent checkpoint on stable storage, reducing storage overhead and eliminating the need for garbage collection. Its main disadvantage, however, is the large latency involved in committing output, since a global checkpoint is needed before messages can be sent to OWP.

A straightforward approach to coordinated checkpointing is to block communications while the checkpointing protocol executes [Tamir and Sequin 1984]. A coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives this message, it stops its execution, flushes all the communication channels, takes a *tentative* checkpoint, and sends an acknowledgment message back to the coordinator. After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol. After receiving the commit message, each process removes the old permanent checkpoint and atomically makes the *tentative* checkpoint permanent. The

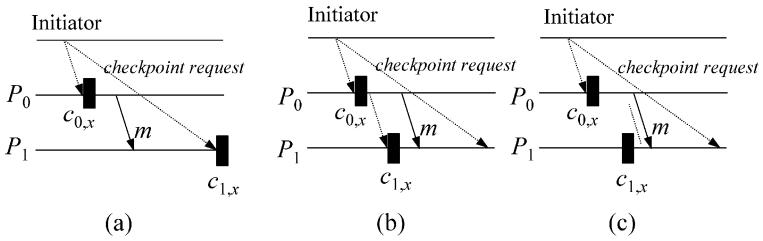

Fig. 8. Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) with FIFO channels; (c) non-FIFO channels (short dashed line represents piggybacked *checkpoint request*).

process is then free to resume execution and exchange messages with other processes. This straightforward approach, however, can result in large overhead, and therefore non-blocking checkpointing schemes are preferable [Elnozahy et al. 1992].

3.2.2. Non-blocking Checkpoint Coordination. A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent. Consider the example in Figure 8(a), in which message m is sent by P_0 *after* receiving a checkpoint request from the checkpoint coordinator. Now, assume that m reaches P_1 *before* the checkpoint request. This situation results in an inconsistent checkpoint since checkpoint $c_{1,x}$ shows the receipt of message m from P_0, while checkpoint $c_{0,x}$ does not show it being sent from P_0. If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, and forcing each process to take a checkpoint upon receiving the first checkpoint-request message, as illustrated in Figure 8(b). An example of a non-blocking checkpoint coordination protocol using this idea is the *distributed snapshot* [Chandy and Lamport 1985], in which *markers* play the role of the checkpoint-request messages. In this protocol, the initiator takes a checkpoint and broadcasts a marker (a checkpoint request) to all processes. Each process takes a checkpoint upon receiving the first marker and rebroadcasts the marker to all processes before sending any application message. The protocol works assuming the channels are reliable and FIFO. If the channels are non-FIFO, the marker can be piggybacked on every post-checkpoint message as in Figure 8(c) [Lai and Yang 1987]. Alternatively, checkpoint indices can serve the same role as markers, where a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked checkpoint index [Elnozahy, et al. 1992; Silva 1997].

3.2.3. Checkpointing with Synchronized Clocks. Loosely synchronized clocks can facilitate checkpoint coordination [Cristian and Jahanian 1991; Tong et al. 1992]. More specifically, loosely synchronized clocks can trigger the local checkpointing actions of all participating processes at approximately the same time without a checkpoint initiator [Cristian and Jahanian 1991]. A process takes a checkpoint and waits for a period that equals the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process in the system. The process can be assured that all checkpoints belonging to the same coordination session have been taken without the need of exchanging any messages. If a failure occurs, it is detected within the specified time and the protocol is aborted.

3.2.4. Checkpointing and Communication Reliability. Depending on the assumption of reliability of the communication channel (Section 2.4), the protocol may require

some messages to be saved as part of the checkpoint. Consider the case where reliable channels are assumed. Suppose process p sends a message m before taking a checkpoint, and that message m arrives at the intended destination at process q after q has taken its checkpoint. In this case, the recorded state of p would show message m to have been sent, while $q's$ state would show that the message has not been received. If a failure were to force p and q to roll back to these checkpoints, it would be impossible to guarantee the reliable delivery of m after recovery. To avoid this problem, the protocol requires that all in-transit messages be saved by their intended destinations as part of their recorded state. However, if reliable channels are not assumed, then in-transit messages need not be saved, as the recorded state in this case would still be consistent with the assumption of the communication channels (in this case, the loss of message m if the system fails and restarts would be equivalent to its loss due to a communication failure in a legal execution).

3.2.5. Minimal Checkpoint Coordination. Coordinated checkpointing requires all processes to participate in every checkpoint. This requirement generates valid concerns about its scalability. It is desirable to reduce the number of processes involved in a coordinated checkpointing session. This can be done since the processes that need to take new checkpoints are only those that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint [Koo and Toueg 1987].

The following two-phase protocol achieves minimal checkpoint coordination [Koo and Toueg 1987]. During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request. Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoints and sends them a request, and so on, until no more processes can be identified. During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes. In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowed.

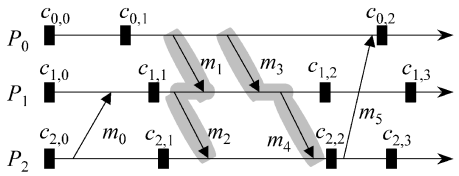

Fig. 9. Z-paths Z cycles.

3.3. Communication-induced Checkpointing

3.3.1. Overview. *Communication-induced checkpointing* (CIC) protocols avoid the domino effect without requiring all checkpoints to be coordinated. In these protocols, processes take two kinds of checkpoints, *local* and *forced*. Local checkpoints can be taken independently, while forced checkpoint *must* be taken to guarantee the eventual progress of the recovery line. In particular, CIC protocols take forced checkpoints to prevent the creation of *useless* checkpoints, that is checkpoints (such as $c_{2,2}$ in Figure 9) that will never be part of a consistent global state. Useless checkpoints are not desirable because they do not contribute to the recovery of the system from failures, but they consume resources and cause performance overhead.

As opposed to coordinated checkpointing, CIC protocols do not exchange any special coordination messages to determine when forced checkpoints should be taken: instead, they piggyback protocol-specific information on each application message; the receiver then uses this information to decide if it should take a forced checkpoint. Informally, this decision is based on the receiver determining if past communication and checkpoint

patterns can lead to the creation of useless checkpoints: a forced checkpoint is then taken to break these patterns. This intuition has been formalized in an elegant theory based on the notions of Z-path and Z-cycle.

A *Z-path* (*zigzag path*) is a special sequence of messages that connects two checkpoints [Netzer and Xu 1995]. Let $\mapsto$ denote Lamport's happen-before relation [Lamport 1978]. Let $c_{i,x}$ denote the x^{th} checkpoint of process P_i. Also, define the execution portion between two consecutive checkpoints on the same process to be the checkpoint interval starting with the earlier checkpoint. Given two checkpoints $c_{i,x}$ and $c_{j,y}$, a Z-path exists between $c_{i,x}$ and $c_{j,y}$ if and only if one of the following two conditions holds:

1. $x < y$ and $i = j$; or
2. There exists a sequence of messages $[m_0, m_1, \ldots, m_n]$, $n \mu 0$, such that:
   - $c_{i,x} \mapsto send_i(m_0)$;
   - $\forall l < n$, either $deliver_k(m_l)$ and $send_k(m_{l+1})$ are in the same checkpoint interval, or $deliver_k(m_l) \mapsto send_k(m_{l+1})$; and
   - $deliver_j(m_n) \mapsto c_{j,y}$

where $send_i$ and $deliver_i$ are communication events executed by process P_i. In Figure 9, $[m_1, m_2]$ and $[m_3, m_4]$ are examples of Z-paths between checkpoints $c_{0,1}$ and $c_{2,2}$.

A *Z-cycle* is a Z-path that begins and ends with the same checkpoint. In Figure 9, the Z-path $[m_5, m_3, m_4]$ is a Z-cycle that starts and ends at checkpoint $c_{2,2}$. Z-cycles are interesting in the context of CIC protocols because it can be proved that a checkpoint is useless if and only if it is part of a Z-cycle [Netzer and Xu 1995]. Hence, one way to avoid useless checkpoints is to make sure that no Z-path ever becomes a Z-cycle.

Traditionally, CIC protocols have been classified in one of two types. *Model-based checkpointing* protocols maintain checkpoint and communication structures that prevent useless checkpoints or achieve some even stronger properties [Wang 1997]. *Index-based coordination* protocols assign timestamps to local and forced checkpoints such that checkpoints with the same timestamp at all processes form a consistent state. Recently, it has been proved that the two types are fundamentally equivalent [Hélary et al. 1997a], although in practice, there may be some evidence that index-based coordination results in fewer forced checkpoints [Alvisi et al. 1999].

3.3.2. Model-based Protocols. Model-based checkpointing relies on preventing patterns of communications and checkpoints that could result in Z-cycles and useless checkpoints. A model is set up to detect the possibility that such patterns could be forming within the system, according to some heuristic. A checkpoint is usually forced to prevent the undesirable patterns from occurring. The decision to force a checkpoint is done locally using the information piggybacked on application messages. Therefore, under this style of checkpointing it is possible that multiple processes detect the potential for inconsistent checkpoints and independently force local checkpoints to prevent the formation of undesirable patterns that may never actually materialize or that could be prevented by a single forced checkpoint. Thus, model-based checkpointing always errs on the conservative side by taking more forced checkpoints than is probably necessary, because without explicit coordination, no process has complete information about the global system state.

The literature contains several domino-effect-free checkpoint and communication models. The *MRS* model [Russell 1980] avoids the domino effect by ensuring that within every checkpoint interval all message-receiving events precede all message-sending events. This model can be maintained by taking an additional checkpoint before every message-receiving event that is not separated from its previous message-sending event by a checkpoint [Wang 1997]. Another way to prevent the domino effect is to avoid rollback propagation completely by taking a checkpoint immediately before every

message-sending event [Bartlett 1981]. Recent work has focused on ensuring that every checkpoint can belong to a consistent global checkpoint and therefore is not useless [Baldoni et al. 1998; Hélary et al. 1997a; Hélary et al. 1997b; Netzer and Xu 1995].

3.3.3. Index-based Protocols. Index-based CIC protocols guarantee, through forced checkpoints if necessary, that (1) if there are two checkpoints $c_{i,m}$ and $c_{j,n}$ such that $c_{i,m} \mapsto c_{j,n}$, then $ts(c_{j,n}) \geq ts(c_{i,m})$, where $ts(c)$ is the timestamp associated with checkpoint c; and (2) consecutive local checkpoints of a process have increasing timestamps. The timestamps are piggybacked on application messages to help receivers decide when they should force a checkpoint. For instance, the protocol by Briatico et al. forces a process to take a checkpoint upon receiving a message with a piggybacked index greater than the local index, and guarantees that the checkpoints having the same index at different processes, form a consistent state [Briatico et al. 1984]. Hélary et al. instead rely on the observation that if checkpoints' timestamps always increase along a Z-path (as opposed to simply non-decreasing, as required by rule (1) above), then no Z-cycle can ever form [Hélary et al. 1997b]. More sophisticated protocols piggyback more information on top of application messages to minimize the number of forced checkpoints [Hélary et al. 1997b].

CIC protocols can potentially have several performance advantages over other styles of checkpointing. Because CIC allows considerable autonomy in deciding when to take checkpoints, processes can take local checkpoints when their state is small and saving it incurs a small overhead [Li and Fuchs 1990; Plank et al. 1995b]. CIC protocols may also, in theory, scale up well in systems with a large number of processes, since they do not require processes to participate in a globally coordinated checkpoint. We discuss the degree to which these advantages materialize in practice in Section 5.

4. LOG-BASED ROLLBACK-RECOVERY

As opposed to checkpoint-based rollback-recovery, log-based rollback-recovery makes explicit use of the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event [Strom and Yemini 1985]. Such an event can be the receipt of a message from another process or an event internal to the process. Sending a message, however, is *not* a nondeterministic event. For example, in Figure 7, the execution of process P_0 is a sequence of four deterministic intervals. The first one starts with the creation of the process, while the remaining three start with the receipt of messages m_0, m_3, and m_7, respectively. Sending message m_2 is uniquely determined by the initial state of P_0 and by the receipt of message m_0, and is therefore not a nondeterministic event.

Log-based rollback-recovery assumes that all nondeterministic events can be identified and their corresponding determinants can be logged to stable storage. During failure-free operation, each process logs the determinants of all the nondeterministic events that it observes onto stable storage. Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery. After a failure occurs, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding nondeterministic events precisely as they occurred during the pre-failure execution. Because execution within each deterministic interval depends only on the sequence of nondeterministic events that preceded the interval's beginning, the pre-failure execution of a failed process can be reconstructed during recovery up to the first nondeterministic event whose determinant is not logged.

Log-based rollback-recovery protocols have been traditionally called "message logging protocols." The association of nondeterministic events with messages is rooted in the earliest systems that proposed and implemented this style of

recovery [Bartlett 1981; Borg, et al. 1989; Strom and Yemini 1985]. These systems translated nondeterministic events into deterministic message-receipt events.

Log-based rollback-recovery protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process, that is, a process whose state depends on a nondeterministic event that cannot be reproduced during recovery. The way in which a specific protocol implements this condition affects the protocol's failure-free performance overhead, latency of output commit, and simplicity of recovery and garbage collection, as well as its potential for rolling back correct processes. There are three flavors of these protocols:

- Pessimistic log-based rollback-recovery protocols guarantee that orphans are never created due to a failure. These protocols simplify recovery, garbage collection and output commit, at the expense of higher failure-free performance overhead.
- Optimistic log-based rollback-recovery protocols reduce the failure-free performance overhead, but allow orphans to be created due to failures. The possibility of having orphans complicates recovery, garbage collection and output commit.
- Causal log-based rollback-recovery protocols attempt to combine the advantages of low performance overhead and fast output commit, but they may require complex recovery and garbage collection.

We present log-based rollback-recovery protocols by first specifying a property that guarantees that no orphans are created during an execution, and then by discussing how the three major classes of log-based rollback-recovery protocols implement this consistency condition.

4.1. The No-Orphans Consistency Condition

Let e be a nondeterministic event that occurs at process p, we define:

- $Depend(e)$, the set of processes that are affected by a nondeterministic event e. This set consists of p, and any process whose state depends on the event e according to Lamport's *happened before* relation [Lamport 1978].
- $Log(e)$, the set of processes that have logged a copy of e's determinant in their volatile memory.
- $Stable(e)$, a predicate that is true if e's determinant is logged on stable storage.

A process p becomes an orphan when p itself does not fail and p's state depends on the execution of a nondeterministic event e whose determinant cannot be recovered from stable storage or from the volatile memory of a surviving process. Formally [Alvisi 1996]:

$$\forall e : \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$$

We call this property the *always-no-orphans* condition. It stipulates that if any surviving process depends on an event e, then either the event is logged on stable storage, or the process has a copy of the determinant of event e. If neither condition is true, then the process is an orphan because it depends on an event e that cannot be generated during recovery since its determinant has been lost.

4.2. Pessimistic Logging

4.2.1. Overview. Pessimistic logging protocols are designed under the assumption that a failure can occur after any nondeterministic event in the computation. This assumption is "pessimistic" since in reality, failures are rare. In their most straightforward form, pessimistic protocols log to stable storage, the determinant of each nondeterministic event before the event is allowed to affect the computation. These pessimistic protocols implement the following property, often referred to as *synchronous logging*, which is a strengthening of the always-no-orphans condition:

$$\forall e : \neg Stable(e) \Rightarrow |Depend(e)| = 0$$

This property stipulates that if an event has not been logged on stable storage, then no process can depend on it.

In addition to logging determinants, processes also take periodic checkpoints to limit the amount of work that has to be repeated in execution replay during recovery. Should a failure occur, the application program is restarted from the most recent checkpoint and the logged determinants are used during recovery to recreate the pre-failure execution.

Consider the example in Figure 10. During failure-free operation the logs of processes P_0, P_1 and P_2 contain the determinants needed to replay messages [m_0, m_4, m_7], [m_1, m_3, m_6] and [m_2, m_5], respectively. Suppose processes P_1 and P_2 fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to again deliver the same sequence of messages as in the pre-failure execution. This guarantees that P_1 and P_2 will exactly repeat their pre-failure execution and re-send the same messages. Hence, once recovery is complete, both processes will be consistent with the state of P_0 that includes the receipt of message m_7 from P_1.

In a pessimistic logging system, the observable state of each process is always recoverable. This property has four advantages:

1. Processes can send messages to the outside world without running a special protocol.
2. Processes restart from their most recent checkpoint upon a failure, therefore limiting the extent of execution that has to be replayed. Thus, the frequency of checkpoints can be determined by trading off the desired runtime performance with the desired protection of the ongoing execution.
3. Recovery is simplified because the effects of a failure are confined only to the processes that fail. Functioning processes continue to operate and never become orphans because a process always recovers to the state that included its most recent interaction with any other process including OWP. This is highly desirable in practical systems [Huang and Wang 1995].
4. Garbage collection is simple. Older checkpoints and determinants of nondeterministic events that occurred before the most recent checkpoint can be reclaimed because they will never be needed for recovery.

The price to be paid for these advantages is a performance penalty incurred by synchronous logging. Implementations of pessimistic logging must therefore resort to special techniques to reduce the effects of synchronous logging on performance. Some protocols rely on special hardware to facilitate logging [Borg, et al. 1989], while others may limit the number of tolerated failures to improve performance [Johnson and Zwaenepoel 1987; Juang and Venkatesan 1991].

4.2.2. Techniques for Reducing Performance Overhead. Synchronous logging can potentially result in a high performance overhead. This overhead can be lowered using special hardware. For example, fast non-volatile semiconductor memory can be used to implement stable storage [Banâtre et al. 1988]. Synchronous logging in such an implementation is orders of magnitude cheaper than with a traditional implementation of stable storage that uses magnetic disk devices. Another form of hardware support uses a special bus to guarantee atomic logging of all messages exchanged in the system [Borg, et al. 1989]. Such hardware support ensures that the log of one machine is automatically stored on a designated backup without blocking the execution of the application program. This scheme, however, requires that all nondeterministic events be converted into *external* messages [Bartlett 1981; Borg, et al. 1989].

Some pessimistic logging systems reduce the overhead of synchronous logging without relying on hardware. For example, the *Sender-Based Message Logging* (SBML) protocol keeps the determinants corresponding to the delivery of each message m in the volatile memory of its sender [Johnson and Zwaenepoel 1987]. The determinant of m, which consists of its content and the order in which it was

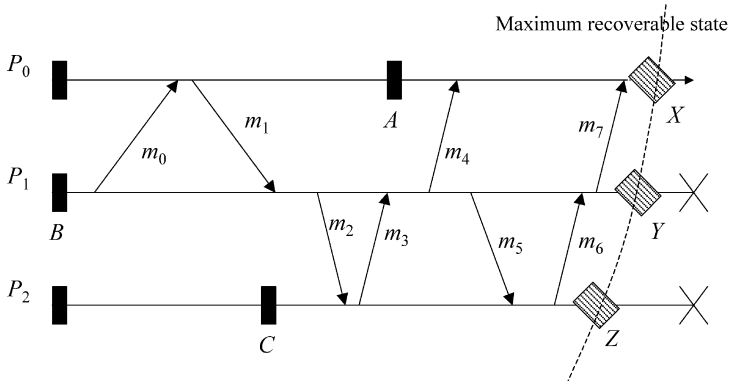

Fig. 10. Pessimistic logging.

delivered, is logged in two steps. First, before sending m, the sender logs its content in volatile memory. Then, when the receiver of m responds with an acknowledgment that includes the order in which the message was delivered, the sender adds the ordering information to the determinant. SBML avoids the overhead of accessing stable storage but tolerates only one failure and cannot handle nondeterministic events internal to a process. Extensions to this technique can tolerate more than one failure in special network topologies [Juang and Venkatesan 1991].

4.2.3. Relaxing Logging Atomicity. The performance overhead of pessimistic logging can be reduced by delivering a message or an event and deferring its logging until the receiver communicates with any other process, including OWP [Johnson and Zwaenepoel 1987]. In the example of Figure 10, process P_0 may defer the logging of messages m_4 and m_7 until it communicates with another process or the outside world. Process P_0 implements the following weaker property, which still guarantees the always-no-orphans condition:

$$\forall e : \neg Stable(e) \Rightarrow |Depend(e)| \leq 1$$

This property relaxes the condition of pessimistic logging by allowing a single process to be affected by an event that has yet to be logged, provided that the process does not externalize the effect of this dependency to other processes including OWP. Thus, messages $m_4$and m_7 are allowed to affect process P_0, but this effect is local—no other process, or the outside world can see it until the messages are logged.

The *observed* behavior of each process is the same as with an implementation that logs events before delivering them to applications. Event logging and delivery are not performed in one atomic operation in this variation of pessimistic logging. This scheme reduces overhead because several events can be logged in one operation, reducing the frequency of synchronous access to stable storage. Latency of interprocess communication and output commit, are not reduced since a logging operation may often be needed before sending a message.

Systems that separate logging of an event from its delivery may lose the last messages delivered before a failure. This may be a problem for applications that assume that processes communicate through reliable channels. Consider one of these applications going through the execution shown in Figure 10, and assume that process P_0 fails after delivering messages m_4 and m_7 but before the corresponding determinants—containing the content and order of receipt of the messages—are logged. Protocols in which the receiver logs the message content cannot guarantee that the recovered P_0 will ever deliver m_4 and m_7, violating the

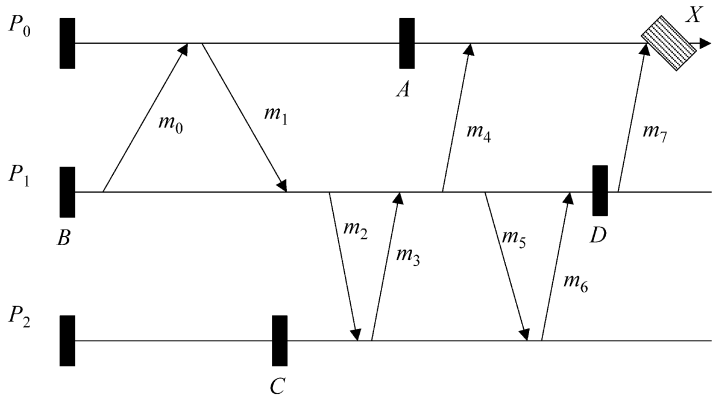

Fig. 11. Optimistic logging.

assumption about reliable channels. This problem does not arise in protocols that log messages at the sender or do not assume reliable communication channels [Elnozahy 1993; Johnson 1989; Johnson and Zwaenepoel 1987].

4.3. Optimistic Logging

4.3.1. Overview. In optimistic logging protocols, processes log determinants *asynchronously* to stable storage [Strom and Yemini 1985]. These protocols make the optimistic assumption that logging will complete before a failure occurs. Determinants are kept in a volatile log, which is periodically flushed to stable storage. Thus, optimistic logging does not require the application to block waiting for the determinants to be actually written to stable storage, and therefore incurs little overhead during failure-free execution. However, this advantage comes at the expense of more complicated recovery and garbage collection, and slower output commit, than in pessimistic logging. If a process fails, the determinants in its volatile log will be lost, and the state intervals that were started by the nondeterministic events corresponding to these determinants cannot be recovered. Furthermore, if the failed process sent a message during any of the state intervals that cannot be recovered, the receiver of the message becomes an orphan process and must roll back to undo the effects of receiving the message. Optimistic protocols do not implement the *always-no-orphans* condition, and therefore permit the temporary creation of orphan processes. However, they require that the property holds by the time recovery is complete. This is achieved during recovery by rolling back orphan processes until their states do not depend on any message whose determinant has been lost. For example, suppose process P_2 in Figure 11 fails before the determinant for m_5 is logged to stable storage. Process P_1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message m_6. The rollback of P_1 further forces P_0 to roll back to undo the effects of receiving message m_7.

To perform these rollbacks correctly, optimistic logging protocols track causal dependencies during failure-free execution. Upon a failure, the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan.

The above example also illustrates why optimistic logging protocols require a nontrivial garbage collection algorithm. While pessimistic protocols need only keep the most recent checkpoint of each process, optimistic protocols may need to keep multiple checkpoints. In the example, the failure of P_2 forces P_1 to restart from checkpoint B instead of its most recent checkpoint D.

Finally, since determinants are logged asynchronously, output commit in optimistic logging protocols generally requires

multi-host coordination to ensure that no failure scenario can revoke the output. For example, if process P_0 needs to commit output at state X, it must log messages m_4 and m_7 to stable storage and ask P_2 to log m_2 and m_5.

4.3.2. Synchronous vs. Asynchronous Recovery. Recovery in optimistic logging protocols can be either *synchronous* or *asynchronous*. In synchronous recovery [Johnson 1989; Sistla and Welch 1989], all processes run a recovery protocol to compute the maximum recoverable system state based on dependency and logged information, and then perform the actual rollbacks. During failure-free execution, each process increments a *state interval index* at the beginning of each state interval. Dependency tracking can be either *direct* or *transitive*.

In direct dependency tracking [Johnson 1989; Sistla and Welch 1989], the state interval index of the sender is piggybacked on each outgoing message to allow the receiver to record the dependency directly caused by the message. These direct dependencies can then be assembled at recovery time to obtain complete dependency information. Alternatively, transitive dependency tracking [Sistla and Welch 1989; Strom and Yemini 1985] can be used: each process P_i maintains a size-N vector TD_i, where $TD_i[i]$ is P_i's current state interval index, and $TD_i[j]$, $j \neq i$, records the highest index of any state interval of P_j on which P_i depends. Transitive dependency tracking generally incurs a higher failure-free overhead for piggybacking and maintaining the dependency vectors, but allows faster output commit and recovery.

In asynchronous recovery, a failed process restarts by sending a rollback announcement broadcast or a recovery message broadcast to start a new *incarnation* [Strom and Yemini 1985]. Upon receiving a rollback announcement, a process rolls back if it detects that it has become an orphan with respect to that announcement, and then broadcasts its own rollback announcement. Since rollback announcements from multiple incarnations of the same process may coexist in the system, each process in general needs to track the dependency of its state on every incarnation of all processes to correctly detect orphaned states. A way to limit dependency tracking to only one incarnation of each process is to force a process to delay its delivery of certain messages. That is, before a process P_i can deliver any message carrying a dependency on an unknown incarnation of process P_j, P_i must first receive rollback announcements from P_j to verify that P_i's current state does not depend on any invalid state of P_j's previous incarnations. Piggybacking all rollback announcements known to a process on every outgoing message can eliminate blocking, and the amount of piggybacked information can be further reduced to a provable minimum [Smith and Johnson 1996].

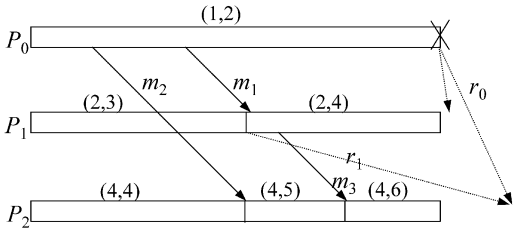

Fig. 12. Exponential rollbacks.

Another issue in asynchronous recovery protocols is the possibility of *exponential rollbacks*. This phenomenon occurs if a single failure causes a process to roll back an exponential number of times [Sistla and Welch 1989]. Figure 12 gives an example, where each integer pair (i, x) represents the xth state interval of the ith incarnation of a process. Suppose P_0 fails and loses its interval (1,2). When P_0's rollback announcement r_0 reaches P_1, the latter rolls back to interval (2,3) and broadcasts another rollback announcement r_1. If r_1 reaches P_2 before r_0 does, P_2 will first roll back to (4,5) in response to r_1, and later roll back again to (4,4) upon receiving r_0. By generalizing this example, we can construct scenarios in which process P_i, $i > 0$, rolls back 2^{i-1} times in response to P_0's failure.

Several approaches have been proposed to ensure that any process will roll back at most once in response to a single failure. Exponential rollbacks can be eliminated by distinguishing failure announcements from rollback announcements and by broadcasting only the former [Sistla and Welch 1989]. Another possibility is to piggyback the original rollback announcement from the failed process on every subsequent rollback announcement that it triggers. For example, in Figure 12, process P_1 piggybacks r_0 on r_1. Exponential rollbacks can be avoided by piggybacking all rollback announcements on every application message [Smith and Johnson 1996].

4.4. Causal Logging

4.4.1. Overview. Causal logging has the failure-free performance advantages of optimistic logging while retaining most of the advantages of pessimistic logging [Alvisi 1996; Elnozahy 1993]. Like optimistic logging, it avoids synchronous access to stable storage except during output commit. Like pessimistic logging, it allows each process to commit output independently and never creates orphans, thereby isolating each process from the effects of failures that occur in other processes. Furthermore, causal logging limits the rollback of any failed process to the most recent checkpoint on stable storage. This reduces the storage overhead and the amount of work at risk. These advantages come at the expense of a more complex recovery protocol.

Causal logging protocols ensure the *always-no-orphans property* by ensuring that the determinant of each nondeterministic event that causally precedes the state of a process is either stable or it is available locally to that process. Consider the example in Figure 13(a). While messages m_5 and m_6 may be lost upon the failure, process P_0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's *happened-before* relation [Lamport 1978]. These events consist of the delivery of messages m_0, m_1, m_2, m_3 and m_4. The determinant of each of these nondeterministic events is either logged on stable storage or is available in the volatile log of process P_0. The determinant of each of these events contains the order in which its original receiver delivered the corresponding message. The message sender, as in sender-based message logging, logs the message content. Thus, process P_0 will be able to "guide" the recovery of P_1 and P_2 since it knows the order in which P_1 should replay messages m_1 and m_3 to reach the state from which P_1 sends message m_4. Similarly, P_0 has the order in which P_2 should replay message m_2 to be consistent with both P_0 and P_1. The content of these messages is obtained from the sender log of P_0 or regenerated deterministically during the recovery of P_1 and P_2. Notice that information about m_5 and m_6 is not available anywhere. These messages may be replayed after recovery in a different order, if at all. However, since they had no effect on a surviving process or the outside world, the resulting state is consistent. The determinant log kept by each process acts as an insurance to protect it from the failures that occur in other processes. It also allows the process to make its state recoverable by simply logging the information available locally. Thus, a process does not need to run a multi-host protocol to commit output.

4.4.2. Tracking Causality. Causal logging protocols implement the *always-no-orphans* condition by having processes piggyback the non-stable determinants in their volatile log on the messages they send to other processes. On receiving a message, a process first adds any piggybacked determinant to its volatile determinant log and then delivers the message to the application.

The *Manetho* system propagates the causal information in an *antecedence graph* [Elnozahy 1993]. The antecedence graph provides every process in the system with a complete history of the nondeterministic events that have causal effects on its state. The graph has a node representing each nondeterministic event that

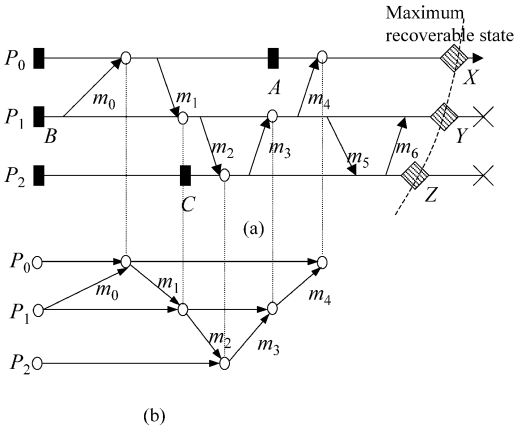

Fig. 13. Causal logging (a) Maximum recoverable states, and (b) antecedence graph of P_0 at state X.

precedes the state of a process, and the edges correspond to the *happened-before* relation [Lamport 1978]. Figure 13(b) shows the antecedence graph of process P_0 of Figure 13(a) at state X. During failure-free operation, each process piggybacks on each application message, the determinants that contain the receipt orders of its direct and transitive antecedents, — its local antecedence graph. The receiver of the message records these receipt orders in its volatile log.

In practice, carrying the entire graph on each application message may lead to an unacceptable overhead. Fortunately, each message carries a graph that is a superset of the one piggybacked on the previous message sent from the same host. This fact can be used in practical implementations to reduce the amount of information carried on application messages. Thus, any message between processes p and q carries only the difference between the graphs piggybacked on that message and the previous message exchanged between these two hosts. Furthermore, if p has recently received a message from q, it can exclude the graph portions that have been piggybacked on that message. Process q already has the information in these excluded portions, therefore transmitting them serves no purpose. Other optimizations are also possible but depend on the semantics of the communication protocol. An implementation of this technique shows that it has very low overhead in practice [Elnozahy 1993].

Further reduction of the overhead is possible if the system is willing to tolerate a number of failures that is less than the total number of processes in the system. This observation is the basis of Family Based Logging protocols (FBL) that are parameterized by the number of tolerated failures [Alvisi 1996]. The basis of these protocols is that, to tolerate f process failures, it is sufficient to log each nondeterministic event in the volatile store of $f+1$ different hosts. Hence, the predicate $Stable(e)$ holds as soon as $|Log(e)| > f$. Sender-based logging is used to support message replay during recovery and determinants are piggybacked on application messages. However, unlike Manetho, propagation of information about an event stops when it has been recorded in $f+1$ processes. For $f < N$, FBL protocols do not access stable storage except for checkpointing. Reducing access to stable storage in turn reduces performance overhead and implementation complexity. Applications pay only the overhead that corresponds to the number of failures they are willing to tolerate. An implementation for the protocol with $f = 1$ confirms that the performance overhead is very small [Alvisi 1996]. The Manetho protocol is an FBL protocol corresponding to the case of $f = N$.

4.5. Comparison

Different rollback-recovery protocols offer different tradeoffs with respect to performance overhead, latency of output commit, storage overhead, ease of garbage collection, simplicity of recovery, freedom from domino effect, freedom from orphan processes, and the extent of rollback. Table I summarizes a comparison among the different variations of rollback-recovery protocols.

Since garbage collection and recovery both involve calculating a recovery line, they can be performed by simple procedures under coordinated checkpointing and pessimistic logging, both of which

Table I. A Comparison Between Various Flavors of Rollback-Recovery Protocols

	Uncoordinated Checkpointing	Coordinated Checkpointing	Comm. Induced Checkpointing	Pessimistic Logging	Optimistic Logging	Causal Logging
PWD assumed?	No	No	No	Yes	Yes	Yes
Checkpoint/ process	Several	1	Several	1	Several	1
Domino effect	Possible	No	No	No	No	No
Orphan processes	Possible	No	Possible	No	Possible	No
Rollback extent	Unbounded	Last global checkpoint	Possibly several checkpoints	Last checkpoint	Possibly several checkpoints	Last checkpoint
Recovery data	Distributed	Distributed	Distributed	Distributed or local	Distributed or local	Distributed
Recovery protocol	Distributed	Distributed	Distributed	Local	Distributed	Distributed
Output commit	Not possible	Global coordination required	Global coordination required	Local decision	Global coordination required	Local decision

have a predetermined recovery line during failure-free execution. The extent of any potential rollback determines the maximum number of checkpoints each process needs to retain. Uncoordinated checkpointing can have unbounded rollbacks, and a process may need to retain up to N checkpoints if the optimal garbage collection algorithm is used [Wang et al. 1995b]. Also, several checkpoints may need to be kept, under optimistic logging, depending on the specifics of the logging scheme. Note that we do not include failure-free overhead as a factor in the comparison. Several studies have shown that these protocols perform reasonably well in practice, and that several factors such as checkpointing frequency, machine speed, and stable storage bandwidth play more important roles than the fundamental aspects of a particular protocol [Alvisi 1996; Elnozahy 1993; Elnozahy, et al. 1992; Huang and Kintala 1993; Johnson 1989; Muller et al. 1994; Plank 1993; Plank, et al. 1995b; Ruffin 1992; Silva 1997].

5. IMPLEMENTATION ISSUES

5.1. Overview

While there is a rich body of research on the algorithmic aspects of rollback-recovery protocols, reports on experimental prototypes or commercial implementations are relatively scarce. The few experimental studies available have shown that building rollback-recovery protocols with low failure-free overhead is feasible. These studies also provide ample evidence that the main difficulty in implementing these protocols lies in the complexity of handling recovery [Elnozahy 1993]. It is interesting to note that all commercial implementations of message logging use pessimistic logging because it simplifies recovery [Borg, et al. 1989; Huang and Wang 1995].

Several recent studies have also challenged some premises on which many rollback-recovery protocols rely. Many of these protocols were introduced in the 1980's, as processor speed and network

bandwidth were such that communication overhead was deemed too high, especially as compared to the cost of stable storage access [Bhargava et al. 1990]. In such platforms, multi-host coordination incurs a large overhead because of the necessary control messages. A protocol that does not require a large communication overhead at the expense of more stable storage accesses performs better in such platforms. Recently, processor speed and network bandwidth have increased dramatically, while the speed of stable storage access has remained relatively the same.[2] This change in the equation suggests a fresh look at the premises of many rollback-recovery protocols. Recent results [Alvisi 1996; Elnozahy 1993; Johnson 1989; Muller, et al. 1994; Plank 1993; Silva 1997; Slye and Elnozahy 1998] have shown that:

- Stable storage access is now the major source of overhead in checkpointing or message logging systems. Communication overhead is much lower in comparison. Such changes favor coordinated checkpointing schemes over message logging or uncoordinated checkpointing systems, as they require less access to stable storage and are simpler to implement.
- The case for message logging has become the ability to interact with the outside world, instead of reducing the overhead of multi-process coordination [Elnozahy and Zwaenepoel 1994]. Message logging systems can implement efficient protocols for committing output and logging input that are not possible in checkpoint-only systems.
- Recent advances have shown that arbitrary forms of nondeterminism can be supported at a very low overhead in logging systems. Nondeterminism was deemed one of the complexities inherent in message logging systems.

In the remainder of this section, we address these and other issues in some detail.

5.2. Checkpointing Implementation

All available studies have shown that writing the state of a process to stable storage is the largest contributor to the performance overhead [Plank 1993]. The simplest way to save the state of a process is to suspend execution, save the process's address space on stable storage, and then resume execution [Tamir and Sequin 1984]. This scheme can be costly for programs with large address spaces if stable storage is implemented using magnetic disks, as it is the custom. Several techniques exist to reduce this overhead.

5.2.1. Concurrent Checkpointing. All available studies show that concurrent checkpointing greatly reduces the overhead of saving the state of a process [Goldberg et al. 1990; Plank 1993]. Concurrent checkpointing relies on the memory protection hardware available in modern computer architectures to continue the execution of the process while its checkpoint is being saved on stable storage. The address space is protected from further modification at the start of a checkpoint and the memory pages are saved to disk concurrently with the program execution. If the program attempts to modify a page, it incurs a protection violation. The checkpointing system copies the page into a separate buffer from which it is saved on stable storage. The original page is unprotected and the application program is allowed to resume. Implementations that do not incorporate concurrent checkpointing may pay a heavy performance overhead unless the checkpointing interval is set to a large value, which in turn would increase the time for rollback.

5.2.2. Incremental Checkpointing. Adding incremental checkpointing [Feldman and

[2] While semiconductor-based stable storage is becoming more widely available, the size-cost ratio is too low compared to disk-based stable storage. It appears that for some time to come, disk-based systems will continue to be the medium of choice for storing the large files that are needed in checkpointing and logging systems.

Brown 1989] to concurrent checkpointing can further reduce the overhead [Elnozahy, et al. 1992]. Incremental checkpointing avoids rewriting portions of the process states that do not change between consecutive checkpoints. It can be implemented by using the dirty-bit of the memory protection hardware or by emulating a dirty-bit in software [Babaoglu and Joy 1981]. A public domain package implementing this technique along with concurrent checkpointing is available [Plank, et al. 1995b].

Incremental checkpointing can also be extended over several processes. In this technique, the system saves the computed parity or some function of the memory pages that are modified across several processes [Plank and Li 1994]. This technique is very similar to parity computation in RAID disk systems. The parity pages can be saved in volatile memory of some other processes thereby avoiding the need to access stable storage. The storage overhead of this method is very low, and it can be adjusted depending on how many failures the system is willing to tolerate.

Another technique for implementing incremental checkpointing is to directly compare the program's state with the previous checkpoint in software, and writing the difference in a new checkpoint [Plank et al. 1995a]. The required storage and computation overhead to perform such a comparison may waste the benefit of incremental checkpointing. Another variation on this technique is to use probabilistic checkpointing [Nam et al. 1997]. The unit of checkpointing in this scheme is a memory block that is typically much smaller than a memory page. Changes to a memory block are detected by computing a signature and comparing it to the corresponding signature in the previous checkpoint. Probabilistic checkpointing is portable, and has lower storage and computation requirements than required by comparing the checkpoints directly. On the downside, computing a signature to detect changes opens the door for aliasing. This problem occurs when the computed signature does not differ from the corresponding one in the previous checkpoint, even though the associated memory block has changed. In such a situation, the memory block is excluded from the new checkpoint, which therefore becomes erroneous. A probabilistic analysis has shown that the likelihood of aliasing in practice is negligible, but an experimental evaluation has shown that probabilistic checkpointing could be unsafe in practice [Elnozahy 1998].

5.2.3. System-level versus User-level Implementations. Support for checkpointing can be implemented in the kernel [Bartlett 1981; Borg, et al. 1989; Elnozahy 1993; Johnson 1989], or it can be implemented by a library linked with the user program [Alvisi 1996; Goldberg, et al. 1990; Huang and Kintala 1993; Plank, et al. 1995b]. Kernel-level implementations are more powerful because they can also capture kernel data structures that support the user process. However, these implementations are necessarily not portable.

Checkpointing can also be implemented in user level. System calls that manipulate memory protection such as *mprotect* of UNIX can emulate concurrent and incremental checkpointing. The *fork* system call of UNIX can implement concurrent checkpointing if the operating system implements *fork* using copy-on-write protection [Goldberg, et al. 1990]. User-level implementations, however, cannot access kernel's data structures that belong to the process, such as open file descriptors and message buffers, but these data structures can be emulated at user level [Huang and Kintala 1993].

5.2.4. Compiler Support. A compiler can be instrumented to generate code that supports checkpointing [Li and Fuchs 1990]. The compiled program contains code that decides when and what to checkpoint. The advantage of this technique is that the compiler can decide on the variables that must be saved, therefore avoiding unnecessary data. For example, dead variables within a program are not saved in a checkpoint though they have been modified. Furthermore, the compiler

may decide the points during program execution where the amount of state to be saved is small.

Despite these promising advantages, there are difficulties with this approach. It is generally undecidable to find the point in program execution most suitable to take a checkpoint. There are, however, several heuristics that can be used. The programmer can provide hints to the compiler about where checkpoints should be inserted or what data variables should be stored [Beguelin et al. 1997; Plank, et al. 1995b]. The compiler may also be trained by running the application in an iterative manner and by observing its behavior [Li and Fuchs 1990]. The observed behavior could help decide the execution points where it would be appropriate to insert checkpoints. Compiler support could also be simplified in languages that support automatic garbage collection [Appel 1989]. The execution point after each major garbage collection provides a convenient place to take a checkpoint at a minimum cost.

5.2.5. Checkpoint Placement. A large amount of work has been devoted to analyzing and deriving the optimal checkpointing frequency and placement [Chandy and Ramamoorthy 1972]. The problem is usually formulated as an optimization problem subject to constraints. For instance, a system may attempt to reduce the number of taken checkpoints subject to a certain limit on the amount of expected rollback. Generally, it has been observed in practice that the overhead of checkpointing is usually negligible unless the checkpointing interval is relatively small, therefore the optimality of checkpoint placement is rarely an issue in practical systems [Elnozahy, et al. 1992].

5.3. Checkpointing Protocols in Comparison

Many checkpointing protocols were introduced at a time when the communication overhead far exceeded the overhead of accessing stable storage. Furthermore, the memory available to run processes tended to be small. These tradeoffs naturally favored uncoordinated checkpointing schemes over coordinated ones. Current technological trends however have reversed this tradeoff.

In modern systems, the overhead of coordinating checkpoints is negligible compared to the overhead of saving the states [Alvisi 1996; Elnozahy 1993; Johnson 1989; Muller, et al. 1994; Plank 1993; Silva 1997]. Using concurrent and incremental checkpointing, the overhead of either coordinated or uncoordinated checkpointing is *essentially the same*. Therefore, uncoordinated checkpointing is not likely to be an attractive technique in practice given the negligible performance gains. These gains do not justify the complexities of finding a consistent recovery line after the failure, the susceptibility to the domino effect, the high storage overhead of saving multiple checkpoints of each process, and the overhead of garbage collection. It follows that coordinated checkpointing is superior to uncoordinated checkpointing when all aspects are considered on balance.

A recent study has also shed some light on the behavior of communication-induced checkpointing [Alvisi, et al. 1999]. It presents an analysis of these protocols based on a prototype implementation and validated simulations, showing that communication-induced checkpointing does not scale well as the number of processes increases. The occurrence of forced checkpoints at random points within the execution due to communication messages makes it very difficult to predict the required amount of stable storage for a particular application run. Also, this unpredictability affects the policy for placing local checkpoints and makes communication-induced protocols cumbersome to use in practice. Furthermore, the study shows that the benefit of autonomy in allowing processes to take local checkpoints at their convenience does not seem to hold. In all experiments, a process takes at least twice as many forced checkpoints as local, autonomous ones.

5.4. Communication Protocols

Rollback-recovery complicates the implementation of protocols used for interprocess communications. Some protocols offer the abstraction of reliable communication channels such as connection-based protocols (e.g. TCP, RPC). Alternatively, other protocols offer the abstraction of an unreliable datagram service (e.g. UDP). Each type of abstraction requires additional support to operate properly across failures and recoveries.

5.4.1. Location-Independent Identities and Redirection. For all communication protocols, a rollback-recovery system must mask the actual identity and location of processes or remote ports from the application program. This masking is necessary to prevent any application program from acquiring a dependency on the location of a certain process, making it impossible to restart the process on a different machine after a failure. A solution to this problem is to assign a logical, location-independent identifier to each process in the system. This scheme also allows the system to appropriately redirect communication to a restarting process after a failure [Elnozahy 1993].

5.4.2. Reliable Channel Protocols. After a failure, identity masking and communication redirection are sufficient for communication protocols that offer the abstraction of an unreliable channel. Protocols that offer the abstraction of reliable channels require additional support. These protocols usually generate a timeout upcall to the application program if the process at the other end of the channel has failed. These timeouts should be masked since the failed program will soon restart and resume computation. If such upcalls are allowed to affect the application, then the abstraction of a reliable system is no longer upheld. The application will have to encode the necessary support to communicate with the failed process after it recovers.

Masking timeouts should also be coupled with the ability of the protocol implementation to reestablish the connection with the restarting process (possibly restarting on a different machine). This support includes the ability to clean up the old connection in an orderly manner, and to establish a new connection with the restarting host. Furthermore, messages retransmitted as part of the execution replay of the remote host must be identified and, if necessary, suppressed. This requires the protocol implementation to include a form of sequence number that is only used for this purpose.

Recovering in-transit messages that are lost because of a failure is another problem for reliable communication protocols. In TCP/IP communication style, for instance, a message is considered delivered once an acknowledgment is received from the remote host. The message itself may linger in the kernel's buffer for a while before the receiving process consumes it. If this process fails, the in-transit messages must be resent to preserve the semantics of the reliable communication channel. Messages must be saved at the sender side for possible retransmission during recovery. This step can be combined in a system that performs sender-based message logging as part of the log maintenance. In other systems that do not log messages or log messages at the receiver, the copying of each message at the sender side introduces overhead and complexity. The complexity is due to the need for executing some garbage collection algorithm with other sites to reclaim the volatile storage.

5.5. Log-based Recovery

5.5.1. Message Logging Overhead. Message logging introduces three sources of overhead. First, each message must in general be copied to the local memory of the process. Second, the volatile log is regularly flushed to stable storage to free up space. Third, message logging nearly doubles the communication bandwidth required to run the application for systems that implement stable storage via a highly available file system accessible through the network. The first source of

overhead may directly affect communication throughput and latency. This is especially true if the copying occurs in the critical path of the inter-process communication protocol. In this respect, sender-based logging is considered more efficient than receiver-based logging because the copying can take place after sending the message over the network. Additionally, the system may combine the logging of messages with the implementation of the communication protocol and share the message log with the transmission buffers. This scheme avoids the extra copying of the message. Logging at the receiver is more expensive because it is in the critical path of the communication protocol.

Another optimization for sender-based logging systems is to use copy-on-write to avoid making extra-copying [Elnozahy and Zwaenepoel 1994]. This scheme works well in systems where broadcast messages are implemented using several point-to-point messages. In this case, copy-on-write will allow the system to have one copy for identical messages and thus reduce the storage and performance overhead of logging. No similar optimization can be performed in receiver-based systems [Elnozahy and Zwaenepoel 1994].

5.5.2. Combining Log-Based Recovery with Coordinated Checkpointing. Log-based recovery has been traditionally presented as a mechanism to allow the use of *uncoordinated* checkpointing with no domino effect. But a system may also combine event logging with coordinated checkpointing, yielding several benefits with respect to performance and simplicity [Elnozahy and Zwaenepoel 1994]. These benefits include those of coordinated checkpointing—such as the simplicity of recovery and garbage collection—and those of log-based recovery—such as fast output commit. Most prominently, this combination obviates the need for flushing the volatile message logs to stable storage in a sender-based logging implementation. Thus, there is no need for maintaining large logs on stable storage, resulting in lower performance overhead and simpler implementations. The combination of coordinated checkpointing and message logging has been shown to outperform one with uncoordinated checkpointing and message logging [Elnozahy and Zwaenepoel 1994]. Therefore, the purpose of logging should no longer be to allow uncoordinated checkpointing. Rather, it should be the desire for enabling fast output commit for those applications that need this feature.

5.6. Stable Storage

Magnetic disks have been the medium of choice for implementing stable storage [Lampson and Sturgis 1979]. Although they are slow, their storage capacity and low cost combination cannot be matched by other alternatives. An implementation of a stable storage abstraction on top of a conventional file system may not be the best choice, however. Such an implementation will not generally give the performance or reliability needed to implement stable storage [Banâtre, et al. 1988; Elnozahy 1993; Ruffin 1992]. Modern file systems tend to be optimized for the pattern of access expected in workstation or personal computing environments. Furthermore, these file systems are often accessed through a network via a protocol that is optimized for small file accesses and not for the large file accesses that are more common in checkpointing and logging.

An implementation of stable storage should bypass the file system layer and access the disk directly. One such implementation is the KitLog package, which offers a log abstraction that can support checkpointing and message logging [Ruffin 1992]. The package runs in conventional UNIX systems and bypasses the file system by accessing the disk in raw mode. There have also been several attempts at implementing stable storage using non-volatile semiconductor memory [Banâtre, et al. 1988]. Such implementations do not have the performance problems associated with disks, but the price and the small storage capacity remain two problems that limit their wide acceptance.

5.7. Support for Nondeterminism

Nondeterminism occurs when the application program interacts with the operating system through system calls and upcalls (asynchronous events). In log-based recovery, these nondeterministic events must be logged on stable storage so that they can be replayed during recovery. Log-based recovery systems differ in the range of actual events that can be covered.

5.7.1. System Calls. System calls in general can be classified into three types [Borg, et al. 1989; Elnozahy 1993; Goldberg, et al. 1990]. Idempotent system calls are those that return deterministic values whenever executed. Examples include calls that return the user identifier of the process owner. These calls do not need to be logged. A second class of calls consists of those that must be logged during failure-free operation but should not be re-executed during execution replay. The results from these calls should simply be replayed to the application program. These calls include those that inquire about the environment, such as getting the current time of day. Re-executing these calls during recovery might return a different value that is inconsistent with the pre-failure execution. The last type of system calls includes those that must be logged during failure-free operation and re-executed during execution replay. These calls generally modify the environment and therefore they must be re-executed to re-establish the environment changes. Examples include calls that allocate memory or create processes. Ensuring that these calls return the same values and generate the same effect during re-execution can be very complex.

5.7.2. Asynchronous Signals. Nondeterminism results from asynchronous signals available in the form of software interrupts under various operating systems. Such signals must be applied at the same execution points during replay to reproduce the same result. Log-based rollback-recovery can cover this form of nondeterminism by taking a checkpoint after the occurrence of each signal, but this can be very expensive [Bartlett 1981]. Alternatively, the system may convert these asynchronous signals to synchronous messages such as in Targon/32 [Borg, et al. 1989], or it may queue the signals until the application polls for them. Both alternatives convert asynchronous event notifications into synchronous ones, which may not be suitable or efficient for many applications. Such solutions also may require substantial modifications to the operating system or the application program.

Another example of nondeterminism that is difficult to track is shared memory manipulation in multithreaded applications. Reconstructing the same execution during replay requires the same interleaving of shared memory accesses by the various threads as in the pre-failure execution. Systems that support this form of nondeterminism supply their own sets of locking primitives, and require applications to use them for protecting access to shared memory [Goldberg, et al. 1990]. The primitives are instrumented to insert an entry in the log identifying the calling thread and the nature of the synchronization operation. However, this technique has several problems. It makes shared memory access expensive, and may generate a large volume of data in the log. Furthermore, if the application does not adhere to the synchronization model (because of a programmer's error, for instance), execution replay may not be possible.

A technique for tracking nondeterminism due to asynchronous signals and interleaved memory access on single processor systems is to use *instruction counters* [Bressoud and Schneider 1995]. An instruction counter is a register that decrements by one upon the execution of each instruction, leading the hardware to generate an exception when the register contents become 0. An instruction counter can thus be used in two modes. In one mode, the register is loaded with the number of instructions to be executed before a breakpoint occurs. After the CPU executes the specified number of instructions, the counter reaches 0 and the hardware

generates an exception. The operating system fields the exception and executes a pre-specified handler. This mode is useful in setting breakpoints efficiently, such as during debugging. In the second mode, the instruction counter is loaded with the maximum value it can hold. Execution proceeds until an event of interest occurs, at which time the content of the counter is sampled, and the number of instructions executed since the time the counter was set is computed and logged. The use of instruction counters has been suggested for debugging shared memory parallel programs [Mellor-Crummey and LeBlanc 1989].

Instruction counters can be used in rollback-recovery to track the number of instructions that occur between asynchronous interrupts [Slye and Elnozahy 1998]. These instruction counts are logged as part of the log that describes the nondeterministic events. During recovery, the system recovers the instruction counts from the log and uses them to regenerate the software interrupts at the same execution points within the application as before the failure. The application therefore goes through the same set of asynchronous events precisely as it did before the failure, and therefore it can reconstruct its execution.

An instruction counter can be implemented in hardware, as in the PA-RISC precision architecture where it has been used to augment the hypervisor of a virtual-machine manager and coordinate a primary virtual machine with its backup [Bressoud and Schneider 1995]. It also can be emulated in software [Mellor-Crummey and LeBlanc 1989]. An implementation study shows that the overhead of program instrumentation and tracking nondeterminism is less than 6% for a variety of user programs and synthetic benchmarks [Slye and Elnozahy 1998].

5.8. Dependency Tracking

Rollback-recovery protocols vary in the ways they track inter-process dependencies. Some protocols require tagging only an index or a sequence number on every application message [Briatico, et al. 1984], while some require the propagation of a vector of timestamps [Strom and Yemini 1985]. At an extreme, some protocols require the propagation of a graph describing the history of the computation [Elnozahy 1993], or matrices containing bit or timestamp vectors [Baldoni, et al. 1998].

Tagging a message with an index or a sequence number on an application message is simple and does not cause any measurable overhead. When dependency tracking, however, requires more complex structures, there are techniques for reducing the amount of actual data that need to be transferred on top of each message. All these techniques revolve around two themes. First, only incremental changes need to be sent. If some elements of a vector or a graph haven't changed since process p last sent a message to process q, then p need only include those elements that have changed. Implementation of this optimization is straightforward in systems that assume FIFO communication channels. When lossy channels are assumed, this optimization is still possible, but at the expense of more processing overhead [Elnozahy 1993].

The other technique for reducing the overhead of dependency tracking exploits the semantics of the applications and the communication patterns [Elnozahy 1993]. For instance, if it can be inferred from the dependency information available to process p that process q already knows parts of the information that is to be piggybacked on a message outgoing to q, then process p can exclude this information. Surprisingly, implementing this optimization is simple and yields good performance [Elnozahy 1993]. Regardless of the particular method used to track inter-process dependencies, various prototype implementations have shown that the overhead resulting from dependency tracking is negligible compared to the overhead of checkpointing or logging [Alvisi 1996; Alvisi, et al. 1999; Bhargava and Lian 1988; Borg, et al. 1989; Elnozahy 1993; Goldberg, et al. 1990; Johnson 1989; Silva 1997].

5.9. Recovery

Handling execution restart and replay is a difficult part of implementing a rollback-recovery system [Borg, et al. 1989]. The major challenge is reintegrating the recovered process in a computation environment that may or may not be the same as the one in which the process was executing before failure.

5.9.1. Reinstating a Process in its Environment. Implanting a process in a different environment during recovery can create difficulties if its state depends on the pre-failure environment. For example, the process may need to access files that exist on the local disk of the machine. The simplest solution to this problem is to attempt to restart the program on the same host. If this is not feasible, then the system must insulate the process from environment-specific variables [Elnozahy 1993]. This can be done for instance by intercepting system calls that return environment-specific results and replacing them with abstract values under the control of the recovery system. Also, file access could be made highly available by placing all files in network-wide highly available file servers or by using dual-ported disks.

Another problem in implementing recovery is the need to reconstruct the auxiliary state within the operating system kernel that supports the recovering process [Elnozahy 1993; Huang and Kintala 1993; Johnson 1989; Plank 1993]. This state is usually outside of the recovery protocol's control during failure-free operation, unless the protocol is implemented inside the operating system. For protocols implemented outside the operating system, the rollback-recovery system must emulate these data structures and log sufficient information to be able to recreate them during recovery. For example, the recovery system may create a data structure to shadow the open file table of a particular process by intercepting all file manipulation calls from the process itself. Then, the recovery system records some information that would enable it to issue requests to the operating system during recovery in order to force the operating system to recreate these data structures indirectly. Obviously, not all states within the operating system kernel can be emulated this way, and therefore, out-of-kernel implementations should have stricter coverage of the system's state that must be emulated. Since most of the applications that benefit from rollback-recovery seem to be in the realm of scientific computing where no sophisticated state is maintained by the kernel on behalf of the processes, this problem does not seem to be severe in that particular context [Plank, et al. 1995b].

5.9.2. Behavior During Recovery. Previous studies have outlined several characteristics of rollback-recovery systems during recovery [Elnozahy 1993; Rao et al. 1998]. For example, it has been observed that for log-based recovery systems, the messages and determinants available in the logs are replayed at a considerably higher speed during recovery than during normal execution. This is because during normal execution, a process may have to block waiting for messages or synchronization events, while during recovery these messages or events can be immediately replayed.

Also, it has been observed that sender-based logging protocols typically slow down recovery if there are multiple failures, because of the need to re-execute some of the processes under control to regenerate the messages. Moreover, some of these protocols may require sympathetic rollbacks [Strom and Yemini 1985], which increase the overhead of synchronizing the processes during recovery. This experimental evidence seems to confirm the tradeoff between protocols that perform well during failure-free executions, such as causal and optimistic logging, and protocols that perform well during recovery, such as pessimistic logging [Rao, et al. 1998]. It is possible to address this tradeoff by performing logging both at the sender and receivers [Strom and Yemini 1985], such that the sender log is volatile and is kept only until the receiver flushes its volatile logs to stable storage.

5.10. Checkpointing and Mobility

Several studies have examined the issues of checkpointing, logging, and rollback-recovery in mobile computing [Prakash and Singhal 1996]. The fundamental concepts of distributed checkpointing, consistency, and rollback are identical to those in traditional distributed systems, but special considerations must be made for issues inherent to mobile computing, such as energy constraints, intermittent communications, and low-performance processors. These issues favor checkpointing protocols that allow maximum autonomy to participating processes, require low overhead in resources, and can function with the minimum possible number of message exchanges. Therefore, independent checkpointing and communication-induced checkpointing tend to be more appropriate for these environments than coordinated checkpointing. Also, log-based recovery protocols that allow a high degree of autonomy during recovery such as receiver-based optimistic or pessimistic logging tend to be more appropriate for these environments than those protocols that require global communication during recovery. Nevertheless, checkpointing and rollback-recovery have yet to prove useful for mobile hosts. The applications in the mobile domain tend to be structured as client-server interactions for which transaction processing on the server is most appropriate. Also, it is often the case for these applications that high availability is more important than fault tolerance or recoverability, favoring some form of replicated server that can continue to function despite a failure of some of its replicas. Finally, there is an emerging generation of handheld devices that are meant to serve as enhanced input-output devices for remote computations, with little processing or storage capacity to support checkpointing or recovery. Whether this situation changes, will depend on whether rollback-recovery proves to be useful outside the scientific and engineering computing domain in which it has proved very successful.

5.11. Rollback-Recovery in Practice

Despite the wealth of research in the area of rollback-recovery in distributed systems, very few commercial systems actually have adopted it. Difficulties in implementing recovery perhaps are the main reason why these protocols have not been widely adopted. Additionally, the range of applications that benefit from these protocols tend to be in the realm of long-running, scientific programs, which are relatively few. Many of these, in fact, are written to run on supercomputers where some facility exists for checkpointing the entire system's state. For the few that run in a distributed system, public domain libraries that implement checkpointing have proved adequate [Plank, et al. 1995b].

Log-based recovery seemed to have less success than checkpoint-only systems. A commercial implementation of pessimistic logging did not fare well, although the reasons are not clear [Borg, et al. 1989]. One could conjecture that the complex modifications made to the operating system and the special-purpose hardware that was used to mitigate performance overhead made the machine expensive. Some other usage of log-based recovery has been reported in telecommunication applications [Huang and Kintala 1993], although there are no reports on how they fared. Interestingly, both commercial implementations used pessimistic logging, and were used for applications where the performance overhead of this form of logging could be tolerated. We are unaware, however, of any use of optimistic or causal logging rollback-recovery protocols in commercial systems.

6. CONCLUDING REMARKS

We have reviewed and compared different approaches to rollback-recovery with respect to a set of properties including the assumption of piecewise determinism, performance overhead, storage overhead, ease of output commit, ease of garbage collection, ease of recovery, freedom from domino effect, freedom from orphan

processes, and the extent of rollback. These approaches fall into two broad categories: checkpointing protocols and log-based recovery protocols.

Checkpointing protocols require the processes to take periodic checkpoints with varying degrees of coordination. At one end of the spectrum, coordinated checkpointing requires the processes to coordinate their checkpoints to form global consistent system states. Coordinated checkpointing generally simplifies recovery and garbage collection, and yields good performance in practice. At the other end of the spectrum, uncoordinated checkpointing does not require the processes to coordinate their checkpoints, but it suffers from potential domino effect, complicates recovery, and still requires coordination to perform output commit or garbage collection. Between these two ends are communication-induced checkpointing schemes that depend on the communication patterns of the applications to trigger checkpoints. These schemes do not suffer from the domino effect and do not require coordination. Recent studies, however, have shown that the nondeterministic nature of these protocols complicates garbage collection and degrades performance.

Log-based rollback-recovery is often a natural choice for applications that frequently interact with the outside world. It allows efficient output commit, and has three flavors, pessimistic, optimistic, and causal. The simplicity of pessimistic logging makes it attractive for practical applications where a high failure-free overhead is tolerable. This form of logging simplifies recovery, output commit, and protects surviving processes from having to roll back. These advantages have made pessimistic logging attractive in commercial environments where simplicity and robustness are necessary. Causal logging reduces the overhead while still preserving the properties of fast output commit and orphan-free recovery. Alternatively, optimistic logging reduces the overhead further at the expense of complicating recovery and increasing the extent of rollback upon a failure.

ACKNOWLEDGMENTS

The authors wish to express their sincere thanks to Pi-Yu Chung, Om Damani, W. Kent Fuchs, Yennun Huang, Chandra Kintala, Andy Lowry, Keith Marzullo, James Plank, Fred Schneider and Paulo Verissimo for valuable discussions, encouragement and comments.

REFERENCES

ALVISI, L. 1996. *Understanding the Message Logging Paradigm for Masking Process Crashes.* Ph.D. Thesis, Cornell University, Department of Computer Science.

ALVISI, L. AND MARZULLO, K. 1998. Message logging: pessimistic, optimistic, causal and optimal. *IEEE Trans. Softw. Eng. 24*, 2, 149–159.

ALVISI, L., ELNOZAHY, E. N., RAO, S., HUSAIN, S. A., and MEL, A. D. 1999. An analysis of communication-induced checkpointing. In *Digest of Papers, FTCS-29, The Twenty Nineth Annual International Symposium on Fault-Tolerant Computing* (Madison, Wisconsin), 242–249.

APPEL, A. W. 1989. *A runtime system*. Technical Report CS-TR220-89, Department of Computer Science, Princeton University.

BABAOGLU, O. AND JOY, W. 1981. Converting a swap-based system to do paging in an architecture lacking page-reference bits. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 78–86.

BALDONI, R., QUAGLIA, F., AND CICIANI, B. 1998. A *VP*-accordant checkpointing protocol preventing useless checkpoints. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 61–67.

BANÂTRE, J. P., BANÂTRE, M., AND MULLER, G. 1988. Ensuring data security and integrity with a fast stable storage. In *Proceedings of The Fourth Conference on Data Engineering*, 285–293.

BARTLETT, J. F. 1981. A Non Stop Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 22–29.

BEGUELIN, A., SELIGMAN, E., AND STEPHAN, P. 1997. Application-level fault tolerance in heterogeneous networks of workstations. *J. Parallel and Distributed Comput. 43*, 2, 147–155.

BHARGAVA, B. AND LIAN, S. R. 1988. Independent checkpointing and concurrent rollback for recovery—An optimistic approach. In *Proceedings, Seventh Symposium on Reliable Distributed Systems*, 3–12.

BHARGAVA, B., LIAN, S. R., AND LEU, P. J. 1990. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proceedings of the Sixth International Conference on Data Engineering*, 182–189.

BORG, A., BLAU, W., GRAETSCH, W., HERMANN, F., AND OBERLE, W. 1989. Fault tolerance

under UNIX. *ACM Trans. Comput. Syst. 7*, 1, 1–24.

BRESSOUD, T. C. AND SCHNEIDER, F. B. 1995. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1–11.

BRIATICO, D., CIUFFOLETTI, A., AND SIMONCINI, L. 1984. A distributed domino-effect free recovery algorithm. In *IEEE International Symposium on Reliability, Distributed Software, and Databases*, 207–215.

CHANDY, M. AND RAMAMOORTHY, C. V. 1972. Rollback and recovery strategies for computer programs. *IEEE Trans. Comput. 21*, 6, 546–556.

CHANDY, M. AND LAMPORT, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 31*, 1, 63–75.

CRISTIAN, F. AND JAHANIAN, F. 1991. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings, Tenth Symposium on Reliable Distributed Systems*, 12–20.

ELNOZAHY, E. N. 1993. *Manetho: Fault Tolerance in Distributed Systems using Rollback-Recovery and Process Replication.* Ph.D. Thesis, Rice University, Department of Computer Science.

ELNOZAHY, E. N. 1998. How safe is probabilistic checkpointing? In *Digest of Papers, FTCS-28, the Twenty Eight Annual International Symposium on Fault-Tolerant Computing*, 358–363.

ELNOZAHY, E. N. AND ZWAENEPOEL, W. 1994. On the use and implementing of message logging. In *Digest of Papers, FTCS-24, The Twenty Fourth International Symposium on Fault-Tolerant Computing*, 298–307.

ELNOZAHY, E. N., JOHNSON, D. B., AND ZWAENEPOEL, W. 1992. The performance of consistent checkpointing. In *Proceedings, Eleventh Symposium on Reliable Distributed Systems*, 39–47.

FELDMAN, S. I. AND BROWN, C. B. 1989. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging 24*, 1, 112–123.

GOLDBERG, A., GOPAL, A., LI, K., STROM, R., AND BACON, D. 1990. Transparent recovery of Mach applications. In *Usenix Mach Workshop Proceedings*, 169–184.

HÉLARY, J. M., MOSTEFAOUI, A., AND RAYNAL, M. 1997a. Virtual precedence in asynchronous systems: concepts and applications. In *Proceedings of the 11th Workshop on Distributed Algorithms, WDAG'97*.

HÉLARY, J. M., MOSTEFAOUI, A., NETZER, R. H., AND RAYNAL, M. 1997b. Preventing useless checkpoints in distributed computations. In *Proceedings, Sixteenth Symposium on Reliable Distributed Systems*, 183–190.

HUANG, Y. AND KINTALA, C. 1993. Software implemented fault tolerance: Technologies and experience. In *Digest of Papers, FTCS-23, the Twenty Third Annual International Symposium on Fault-Tolerant Computing*, 2–9.

HUANG, Y. AND WANG, Y.-M. 1995. Why optimistic message logging has not been used in telecommunication systems. In *Digest of Papers, FTCS-25, the Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, 459–463.

JOHNSON, D. B. 1989. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. Ph.D. Thesis, Rice University, Department of Computer Science.

JOHNSON, D. B. AND ZWAENEPOEL, W. 1987. Sender-based message logging. In *Digest of Papers, FTCS-17, The Seventeenth Annual International Symposium on Fault-Tolerant Computing*, 14–19.

JOHNSON, D. B. AND ZWAENEPOEL, W. 1990. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms 11*, 3, 462–491.

JUANG, T. T.-Y. AND VENKATESAN, S. 1991. Crash recovery with little overhead. In *Proceedings, The 11th International Conference on Distributed Computing Systems*, 454–461.

KOO, R. AND TOUEG, S. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Engin. 13*, 1, 23–31.

LAI, T. H. AND YANG, T. H. 1987. On distributed snapshots. *Information Processing Letters 25*, 153–158.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7, 588–565.

LAMPSON, B. W. AND STURGIS, H. E. 1979. *Crash recovery in a distributed data storage system*. Technical Report, Xerox Palo Alto Research Center.

LI, C. C. AND FUCHS, W. K. 1990. CATCH: Compiler-assisted techniques for checkpointing. In *Digest of Papers, FTCS-20, The Twentieth Annual International Symposium on Fault-Tolerant Computing*, 74–81.

MELLOR-CRUMMEY, J. AND LEBLANC, T. 1989. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 78–86.

MORIN, C. AND PUAUT, T. 1997. A survey of recoverable distributed shared memory systems. *IEEE Trans. Parallel and Distributed Syst. 8*, 9, 959–969.

MULLER, G., HUE, M., AND PEYROUZ, N. 1994. Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment. In *Lecture Notes in Computer Science: Dependable Computing, EDDC-1*, 491–508.

NAM, H.-C., KIM, J., HONG, S. J., AND LEE, S. 1997. Probabilistic checkpointing. In *Digest of Papers, FTCS-27, The Twenty Seventh Annual International Symposium on Fault-Tolerant Computing*, 48–57.

NETZER, R. H. AND XU, J. 1995. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel and Distributed Syst. 6*, 2, 165–169.

PAUSCH, R. 1988. *Adding Input and Output to the Transactional Model*. Ph.D. Thesis, Carnegie Mellon University, Department of Computer Science.

PLANK, J. S. 1993. *Efficient Checkpointing on MIMD Architectures*. Ph.D. Thesis, Princeton University, Department of Computer Science.

PLANK, J. S. AND LI, K. 1994. Faster checkpointing with $N+1$ parity. In *Digest of Papers, FTCS-24, The Twenty Fourth Annual International Symposium on Fault-Tolerant Computing*, 288–297.

PLANK, J. S., XU, J., AND NETZER, R. H. 1995a. *Compressed differences: An algorithm for fast incremental checkpointing*. Technical Report CS-95-302, University of Tennessee at Knoxville.

PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. 1995b. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USENIX Winter 1995 Technical Conference*, 213–223.

PRAKASH, R. AND SINGHAL, M. 1996. Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Trans. Parallel and Distributed Syst. 7*, 10, 1035–1048.

RANDELL, B. 1975. System structure for software fault tolerance. *IEEE Trans. Softw. Engin. 1*, 2, 220–232.

RAO, S., ALVISI, L., AND VIN, H. M. 1998. The cost of recovery in message logging protocols. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 10–18.

RUFFIN, M. 1992. KITLOG: A generic logging service. In *Proceedings, Eleventh Symposium on Reliable Distributed Systems*, 139–148.

RUSSELL, D. L. 1980. State restoration in systems of communicating processes. *IEEE Trans. Softw. Engin. 6*, 2, 183–194.

SCHLICHTING, R. D. AND SCHNEIDER, F. B. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3, 222–238.

SILVA, L. M. 1997. *Checkpointing Mechanisms for Scientific Parallel Applications*. Ph.D. Thesis, University of Coimbra, Department of Computer Science.

SISTLA, A. AND WELCH, J. 1989. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (*PODC*), 223–238.

SLYE, J. H. AND ELNOZAHY, E. N. 1998. Support for software interrupts in log-based rollback-recovery. *IEEE Trans. Comput. 47*, 10, 1113–1123.

SMITH, S. W. AND JOHNSON, D. B. 1996. Minimizing timestamp size for completely asynchronous optimistic recovery with minimal rollback. In *Proceedings, the Fifteenth Symposium on Reliable Distributed Systems*, 66–75.

STROM, R. AND YEMINI, S. 1985. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst. 3*, 3, 204–226.

TAMIR, Y. AND SEQUIN, C. H. 1984. Error recovery in multicomputers using global checkpoints. In *Proceedings of the International Conference on Parallel Processing*, 32–41.

TONG, Z., KAIN, R. Y., AND TSAI, W. T. 1992. Rollback-recovery in distributed systems using loosely synchronized clocks. *IEEE Trans. Parallel and Distributed Syst. 3*, 2, 246–251.

WANG, Y.-M. 1993. *Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems*. Ph.D. Thesis, University of Illinois, Department of Computer Science.

WANG, Y.-M. 1997. Consistent global checkpoints that contain a set of local checkpoints. *IEEE Trans. Comput. 46*, 4, 456–468.

WANG, Y.-M., CHUNG, P. Y., AND FUCHS, W. K. 1995a. *Tight upper bound on useful distributed system checkpoints*. Technical Report, University of Illinois.

WANG, Y.-M., CHUNG, P. Y., LIN, I. J., AND FUCHS, W. K. 1995b. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. Parallel and Distributed Syst. 6*, 5, 546–554.

Received October 1996; revised June 1997, June 1999, February 2001; accepted April 2002

Rollback-Recovery Protocols in Message-Passing Systems

BIBLIOGRAPHY

ACHARYA, A. AND BADRINATH, B. R. 1992. Recording distributed snapshots based on causal order of message delivery. *Information Processing Letters 44*, 6.

ACHARYA, A. AND BADRINATH, B. R. 1994. Checkpointing distributed applications on mobile computers. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*.

AHAMAD, M. AND LIN, L. 1989. Using checkpoints to localize the effects of faults in distributed systems. In *Proceedings, Eighth Symposium on Reliable Distributed Systems*, 2–11.

AHUJA, M. 1989. *Repeated global snapshots in asynchronous distributed systems*. Technical Report OSU-CISRC-8/89 TR40, The Ohio State University.

ALGUDADY, M. S. AND DAS, C. R. 1991. A cache-based checkpointing scheme for MIN-based multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, 497–500.

ALVISI, L. AND MARZULLO, K. 1995. Message logging: Pessimistic, optimistic and causal. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (Vancouver, Canada).

ALVISI, L. AND MARZULLO, K. 1995. Deriving optimal checkpointing protocols for distributed shared memory architectures. In *Proceedings of the 1995 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)* (Ottawa, Canada).

ALVISI, L. AND MARZULLO, K. 1996. Tradeoffs in implementing causal message logging protocols. In *Proceedings of the 1996 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 58–67.

ALVISI, L., HOPPE, B., AND MARZULLO, K. 1993. Nonblocking and orphan-free message logging protocols. In *Digest of Papers, FTCS-23, The Twenty Third Annual International Symposium on Fault-Tolerant Computing* (Toulouse, France), 145–154.

ALVISI, L., RAO, S., AND VIN, H. M. 1998. Low-overhead protocols for fault-tolerant file sharing. In *Proceedings of the IEEE 18th International Conference on Distributed Computing Systems*, 452–461.

ALVISI, L., BHATIA, K., AND MARZULLO, K. 2000. *Tracking causality in causal message logging protocols*. Technical Report, The University of Texas at Austin.

ALVISI, L., BRESSOUD, T. C., EL-KHASHAB, A., MARZULLO, K., AND ZAGORODNOV, D. 2001. Wrapping server-side TCP to mask connection failures. In *InfoComm*.

ANYANWU, J. A. 1985. A reliable stable storage system for UNIX. *Software–Practice and Experience 15*, 10, 973–900.

ARTSY, Y. AND FINKEL, R. 1989. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 47–56.

ATTIG, N. AND SANDER, V. 1993. Automatic checkpointing of NQS batch jobs on CRAY UNICOS. In *Proceedings of the Cray User Group Meeting*.

BABAOGLU, O. 1990. Fault-tolerant computing based on Mach. In *Proceedings of the USENIX Mach Symposium*, 186–199.

BABAOGLU, O. AND MARZULLO, K. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Mullender, S. ed. *Distributed Systems*, Addison-Wesley, 55–96.

BACON, D. 1991. Transparent recovery in distributed systems. *Operating Systems Review*, 91–94.

BACON, D. 1991. File system measurements and their application to the design of efficient operation logging algorithms. In *Proceedings, Tenth Symposium on Reliable Distributed Systems*, 21–30.

BALDONI, R., QUAGLIA, F., AND FORNARA, P. 1997. An index-based checkpointing algorithm for autonomous distributed systems. In *Proceedings, Sixteenth Symposium on Reliable Distributed Systems,* 27–34.

BALDONI, R., HÉLARY, J.-M., MOSTEFAOUI, A., AND RAYNAL, M. 1997. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Digest of Papers, (FTCS-27), The Twenty Seventh Annual International Symposium on Fault-Tolerant Computing* (Seattle), 68–77.

BALDONI, R., HÉLARY, J.-M., MOSTEFAOUI, A., AND RAYNAL, M. 1997. Adaptive checkpointing in message passing distributed systems. *International Journal of Systems Science 28*, 11, 1145–1161.

BALDONI, R., HÉLARY, J.-M., MOSTEFAOUI, A., AND RAYNAL, M. 1997. Adaptive checkpointing in message passing distributed systems. *International Journal of Systems Science 28*, 11, 1145–1161.

BANÂTRE, J. P., BANÂTRE, M., AND MULLER, G. 1989. Architecture of fault-tolerant multiprocessor workstations. In *Proceedings of the Workshop on Workstation Operating Systems*, 20–24.

BANÂTRE, M., HENG, P., MULLER, G., AND ROCHARD, B. 1991. How to design reliable servers using fault-tolerant micro-kernel mechanisms.

In *Proceedings of the USENIX Mach Symposium*, 223–231.

BANÂTRE, M., GEFFLAUT, A., JOUBERT, P., LEE, P., AND MORIN, C. 1993. *An architecture for tolerating processor failures in shared-memory multiprocessors*. Technical Report No. 707–93, IRISA.

BARIGAZZI, G. AND STRIGINI, L. 1983. Application-transparent setting of recovery points. In *Digest of Papers, FTCS-13, The Thirteenth Annual International Symposium on Fault-Tolerant Computing*, 48–55.

BECK, M., PLANK, J. S., AND KINGSLEY, G. 1994. *Compiler-assisted checkpointing*. Technical Report CS-94-269, University of Tennessee at Knoxville, Department of Computer Science.

BEEDUBAIL, G., KARMARKAR, A., GURIJALA, A., MARTI, W., AND POOCH, U. 1995. An algorithm for supporting fault-tolerant objects in distributed object oriented operating systems. In *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)*, 142–148.

BHATIA, K., MARZULLO, K., AND ALVISI, L. 1998. The relative overhead of piggybacking in causal message logging protocols. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 348–353.

BIEKER, B., DEONINCK, G., MAEHLE, E., AND VOUNCKX, J. 1994. Reconfiguration and checkpointing in massively parallel systems. In *Proceedings of the 1st European Dependant Computing Conference (EDCC-1)*, 353–370.

BORG, A., BAUMBACH, J., AND GLAZER, S. 1983. A message system supporting fault tolerance. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, 90–99.

BOWEN, N. S., AND PRADHAN, D. K. 1991. *Survey of checkpoint and rollback-recovery techniques*. Technical Report TR-91-CSE-17, Department of Electrical and Computer Engineering, Univ. of Mass.

BOWEN, N. S. AND PRADHAN, D. K. 1992. Virtual checkpoints: Architecture and performance. *IEEE Transactions on Computers 41*, 5, 516–525.

BOWEN, N. S. AND PRADHAN, D. K. 1993. Processor- and memory-based checkpoint and rollback-recovery. *IEEE Computer 26*, 2, 22–32.

CABILLIC, G., MULLER, G., AND PUAUT, I. 1995. The performance of consistent checkpointing in distributed shared memory systems. In *Proceedings, Fourteenth Symposium on Reliable Distributed Systems*.

CAMPOS, A. E. AND CASTILLO, M. A. 1996. *Checkpointing through garbage collection*. Technical Report, Escuela de Ingeniería Pontificia Universidad Católica de Chile, Departamento de Ciencia de la Computación.

CAO, J. 1991. On correctness of distributed rollback-recovery. In *Proceedings of the 14th Australia Computer Science Conference*, 39. 31–39.10.

CAO, J. 1992. Efficient synchronous checkpointing in distributed systems. In *Proceedings of the 15th Australia Computer Science Conference*, 165–179.

CAO, J. AND WANG, K. C. 1991. *Efficient synchronous checkpointing in distributed systems*. Technical Report 91/6, James Cook University of North Queensland, Department of Computer Science.

CAO, J. AND WANG, K. C. 1992. An abstract model of rollback-recovery control in distributed systems. *Operating Systems Review*, 62–76.

CAO, G. AND SINGHAL, M. 1998. On the impossibility of min-process non-blocking checkpointing and an efficient checkpointing algorithm for mobile computing systems. In *Proceedings.1998 International Conference on Parallel Processing*, 37–44.

CAO, G. AND SINGHAL, M. 1998. Low-cost checkpointing with mutable checkpoints in mobile computing systems. In *Proceedings of the 18th International Conference on Distributed Computing*, 464–471.

CARGILL, T. AND LOCANTHI, B. 1987. Cheap hardware support for software debugging and profiling. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, 82–83.

CARTER, J. B., COX, A., DWARKADAS, S., ELNOZAHY, E. N., JOHNSON, D. B., KELEHER, P., RODRIGUES, S., YU, W., AND ZWAENEPOEL, W. 1993. Network multicomputing using recoverable distributed shared memory. In *Proceedings of COMPCON'93*.

CASAS, J., CLARK, D., GALBIATI, P., AND KONURU, R. 1995. *MIST: PVM with transparent migration and checkpointing*. Technical Report, Oregon Graduate Institute of Science and Technology, Department of Computer Science.

CHEN, R. AND NG, T. P. 1990. Building a fault-tolerant system based on Mach. In *Proceedings of the USENIX Mach Workshop*, 157–168.

CHEN, R. AND NG, T. P. 1992. Microkernel support for checkpointing. In *Open Forum*.

CHIU, G.-M. AND YOUNG, C.-R. 1996. Efficient rollback-recovery technique in distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems 7*, 6.

CHIUEH, T. 1992. Polar: A storage architecture for fast checkpointing. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, 251–258.

CHIUEH, T.-C. AND DENG, P. 1996. Evaluation of checkpoint mechanisms for massively parallel machines. In *Digest of Papers, FTCS-26, The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, 370–379.

CHOY, M., LEONG, H., AND WONG, M. H. 1995. On distributed object checkpointing and recovery.

In *Proceedings of the 1995 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*.

CHUNG, K.-S., KIM, K.-B., HWANG, C.-S., SHON, J. G., AND YU, H.-C. 1997. Hybrid checkpointing protocol based on selective sender-based message logging. In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, 788–793.

CLEMATIS, A. 1994. Fault-tolerant programming for network based parallel computing. *Microprocessing and Microprogramming 40*, 765–768.

CLEMATIS, A., DODERO, G., AND GIANUZZI, V. 1992. Process checkpointing primitives for fault tolerance: Definitions and examples. *Microprocessors and Microsystems 16*, 1, 15–23.

CUMMINGS, D. AND ALKALAJ, L. 1994. Checkpoint/rollback in a distributed system using coarse-grained dataflow. In *Digest of Papers, FTCS-24, The Twenty Fourth Annual International Symposium on Fault-Tolerant Computing*, 424–433.

DAMANI, O. P. AND GARG, V. K. 1996. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing*, 108–115.

DECONINCK, G. AND LAUWEREINS, R. 1997. User-triggered checkpointing: system-independent and scalable application recovery. In *Proceedings Second IEEE Symposium on Computer and Communications*, 418–423.

DECONINCK, G., VOUNCKX, J., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1993. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. In *IASTED International Conference on Modeling and Simulation*, 262–265.

DECONINCK, G., VOUNCKX, J., LAUWEREINS, R., AND PEPERSTRAETE, J. 1998. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. *International Journal of Modeling and Simulation 18*, 1, 66–71.

DI, Z. 1987. Eliminating domino effect in backward error recovery in distributed systems. In *Proceedings of the 2nd International Conference on Computers and Applications*, 243–248.

DIETER, W. AND LUMPP JR., J. 1999. A user-level checkpointing library for POSIX threads programs. In *Digest of Papers, FTCS-29, The Twenty Ninth Annual International Symposium on Fault-Tolerant Computing*, 224–227.

DUDA, A. 1983. The effects of checkpointing on program execution time. *Information Processing Letters 16*, 221–229.

ECUYER, P. L. AND MALEFANT, J. 1988. Computing optimal checkpointing strategies for rollback and recovery systems. *IEEE Transactions on Computers 37*, 491–496.

ELLENBERGER, E. L. 1995. *Transparent process rollback-recovery: Some new techniques and a portable implementation*. Technical Report, Texas A&M University, Department of Computer Science.

ELLIS, B. 1985. A stable storage package. In *Proceedings of the USENIX Summer Technical Conference*, 209–212.

ELNOZAHY, E. N. 1990. *Efficient fault-tolerance support for interactive distributed applications*. Technical Report TR90-120, Rice University, Department of Computer Science.

ELNOZAHY, E. N. 1994. Fault tolerance for clusters of workstations. In Banâtre, M. AND Lee, P. eds. *Hardware and software architectures for fault tolerance*, Springer Verlag.

ELNOZAHY, E. N. 1995. On the relevance of communication costs of rollback-recovery protocols. In *Proceedings of the 1995 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*.

ELNOZAHY, E. N. AND ZWAENEPOEL, W. 1992. Manetho, transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing 41*, 5, 526–531.

ELNOZAHY, E. N. AND ZWAENEPOEL, W. 1992. Replicated distributed processes in Manetho. In *Digest of Papers, FTCS-22, The Twenty Second Annual International Symposium on Fault-Tolerant Computing*, 18–27.

ELNOZAHY, E. N. AND ZWAENEPOEL, W. 1992. An integrated approach to fault tolerance. In *Proceedings of the Second Workshop on Management of Replicated Data*, 82–85.

FIDGE, C. J. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 55–66.

FISCHER, M. J., GRIFFETH, N. D., AND LYNCH, N. A. 1982. Global states of a distributed system. *IEEE Transactions on Software Engineering SE-8*, 3, 198–202.

FRAZIER, T. M. AND TAMIR, Y. 1989. Application-transparent error-recovery techniques for multicomputers. In *Proceedings of The Fourth Conferences on Hypercubes, Concurrent computers, and Applications*, 103–108.

GAIT, J. 1990. A checkpointing page store for write-once optical disk. *IEEE Transactions on Computers 39*, 1, 2–9.

GARG, S. AND WONG, K. F. 1993. Improving the speed of a distributed checkpointing algorithm. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing Systems*.

GELENBE, E. 1979. On the optimum checkpointing interval. *Journal of the ACM 2*, 259–270.

GELENBE, E. AND DEROCHETTE, D. 1978. Performance of rollback-recovery systems under intermittent failures. *Communications of the ACM 21*, 6, 493–499.

GOLDING III, R. AND SINGHAL, M. 1993. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings, Twelfth Symposium on Reliable Distributed Systems*, 58–67.

GREGORY, S. T. AND KNIGHT, J. C. 1989. On the provision of backward error recovery in production programming languages. In *Digest of Papers, FTCS-19, The Nineteenth Annual International Symposium on Fault-Tolerant Computing*, 506–511.

GROSELJ, B. 1993. Bounded and minimum global snapshots. *IEEE Parallel and Distributed Technology 1*, 4.

HADZILACOS, V. 1982. An algorithm for minimizing rollback cost. In *Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems*, 93–97.

HÉLARY, J. M. 1989. Observing global states of asynchronous distributed applications. *Lecture Notes in Computer Science 392*, 124–135.

HÉLARY, J. M., MOSTEFAOUI, A., AND RAYNAL, M. 1998. Communication-induced determination of consistent snapshots. In *Digest of Papers, FTCS-28, The Twenty Eighth Annual International Symposium on Fault-Tolerant Computing*, 208–217.

HÉLARY, J. M., MOSTEFAOUI, A., AND RAYNAL, M. 1999. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems 10*, 9, 865–877.

HEWITT, C. E. 1980. *Checkpoint and recovery in ACTOR systems*. Technical Report, MIT, Artificial Intelligence Laboratory.

HIGAKI, H. AND TAKIZAWA, M. 1998. Checkpoint-recovery protocol for reliable mobile systems. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 93–99.

HIGAKI, H., SHIMA, K., TACHIKAWA, T., AND TAKIZAWA, M. 1997. Checkpoint and rollback in asynchronous distributed systems. In *Proceedings IEEE INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 998–1005.

ISRAEL, S. AND MORRIS, D. 1989. A non-intrusive checkpointing protocol. In *Proceedings of the Phoenix Conference on Communications and Computers*, 413–421.

JALOTE, P. 1989. Fault-tolerant processes. *Distributed Computing 3*, 187–195.

JANAKIRAMAN, G. AND TAMIR, Y. 1994. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Proceedings, Thirteenth Symposium on Reliable Distributed Systems*, 42–51.

JANSSENS, B. AND FUCHS, W. K. 1993. Relaxing consistency in recoverable distributed shared memory. In *Digest of Papers, FTCS-23, The Twenty Third Annual International Symposium on Fault-Tolerant Computing*, 155–163.

JANSSENS, B. AND FUCHS, W. K. 1994. Reducing interprocessor dependence in recoverable distributed shared memory. In *Proceedings, Thirteenth Symposium on Reliable Distributed Systems*, 34–41.

JASPER, D. P. 1969. A discussion of checkpoint restart. *Software Age*.

JEONG, K. AND SHASHA, D. 1994. Plinda 2.0: A transactional/checkpoint approach to fault-tolerant Linda. In *Proceedings, Thirteenth Symposium on Reliable Distributed Systems*, 96–105.

JOHNSON, D. B. 1993. Efficient transparent optimistic rollback-recovery for distributed application programs. In *Proceedings, Twelfth Symposium on Reliable Distributed Systems*, 86–95.

JOHNSON, D. B. AND ZWAENEPOEL, W. 1988. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC-88)*, 171–181.

JOHNSON, D. B. AND ZWAENEPOEL, W. 1990. *Output-driven distributed optimistic message logging and checkpointing*. Technical Report TR90-118, Rice University, Department of Computer Science.

JOHNSON, D. B., AND ZWAENEPOEL, W. 1991. Transparent optimistic rollback-recovery. *Operating Systems Review*, 99–102.

KAASHOEK, M. F., MICHIELS, R., BAL, H. E., AND TANENBAUM, A. S. 1992. Transparent fault-tolerance in parallel Orca programs. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, 297–312.

KAMBHALTA, S. AND WALPOLE, J. 1990. Recovery with limited replay: Fault-tolerant processes in Linda. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, 715–718.

KANT, K. 1978. A model for error recovery with global checkpointing. *Information Sciences 30*, 58–68.

KANTHADAI, S. AND WELCH, J. L. 1996. Implementation of recoverable distributed shared memory by logging writes. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, 27–30.

KERMARREK, A. M., CABILLIC, G., GEFFLAUT, A., MORIN, C., AND PUAUT, I. 1995. A recoverable distributed shared memory integrating coherence and recoverability. In *Digest of Papers, FTCS-25, The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, 289–298.

KIM, K. H. 1982. Approaches to mechanization of the conversation scheme based on monitors. *IEEE Transactions on Software Engineering SE-8*, 3, 189–197.

KIM, K. H. 1988. Programmer-transparent coordination of recovering concurrent processes: Philosophy and rules for efficient implementation. *IEEE Transactions on Software Engineering SE-14*, 6, 189–197.

KIM, K. H. AND YOU, J. H. 1990. A highly decentralized implementation model for the Programmer-Transparent Coordination (PTC) scheme for cooperative recovery. In *Digest of Papers, FTCS-20, The Twentieth Annual International Symposium on Fault-Tolerant Computing*, 282–289.

KIM, J. L. AND PARK, T. 1993. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems 4*, 8, 955–960.

KIM, K. H., YOU, J. H., AND ABOUELNAGA, A. 1986. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Digest of Papers, FTCS-16, The Sixteenth Annual International Symposium on Fault-Tolerant Computing*, 130–135.

KIM, Y., PLANK, J. S., AND DONGARRA, J. J. 1996. Fault-tolerant matrix operations using checksum and reverse computation. In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation*.

KIM, Y., PLANK, J. S., AND DONGARRA, J. J. 1997. Fault-tolerant matrix operations for network of workstations using multiple checkpointing. In *Proceedings of HPC Asia'97, High Performance Computing in the Information Superhighway*, 460–465.

KINGSBURY, B. A. AND KLINE, J. T. 1989. Job and process recovery in a UNIX-based operating system. In *Usenix Association Winter Conference Proceedings*, 355–364.

KLAIBER, A. C. AND LEVY, H. M. 1993. Crash recovery for scientific applications. In *Proceedings of the International Conference on Parallel and Distributed Systems*.

KRISHNA, P., VAIDYA, N. T., AND PRADHAN, D. K. Recovery in distributed mobile environments. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems* (Princeton, New Jersey), 83–88.

KRISHNA, C. M., KANG, G., AND LEE, Y. 1984. Optimization criteria for checkpoint placement. *Communications of the ACM 27*, 10, 1008–1012.

KRISHNA, P., VAIDYA, N. T., AND PRADHAN, D. K. 1994. Recovery in multicomputers with finite error detection latency. In *Proceedings of the 23rd International Conference on Parallel Processing*.

LAI, T. H. AND YANG, T. H. 1987. On distributed snapshots. *Information Processing Letters 25*, 153–158.

LAMPORT, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems 6*, 2, 254–280.

LANDAU, C. R. 1992. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*.

LEE, B., PARK, T., YEOM, H., AND CHO, Y. 1998. An efficient algorithm for causal message logging. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 19–25.

LEON, J., FICHER, A. L., AND STEENKISTE, P. 1993. *Fail-safe PVM: A portable package for distributed programming with transparent recovery*. Technical Report CMU-CS-93-124, Carnegie Mellon University, School of Computer Science.

LEONG, H. V. AND AGRAWAL, D. 1994. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems (ICDCS-13)*, 227–234.

LEU, P.-J. AND BHARGAVA, B. 1988. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the International Conference on Data Engineering*, 154–163.

LI, W.-J. AND TSAY, J.-J. 1997. Checkpointing message-passing interface (MPI) parallel programs. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, 147–152.

LI, K., NAUGHTON, J. F., AND PLANK, J. S. 1990. Real-time concurrent checkpoint for parallel programs. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, 79–88.

LI, K., NAUGHTON, J. F., AND PLANK, J. S. 1991. Checkpointing multicomputer applications. In *Proceedings, Tenth Symposium on Reliable Distributed Systems*, 1–10.

LI, K., NAUGHTON, J. F., AND PLANK, J. S. 1992. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Programming 20*, 3, 159–180.

LIN, L. AND AHAMAD, M. 1990. Checkpointing and rollback-recovery in distributed object based systems. In *Digest of Papers, FTCS-20, The Twentieth Annual International Symposium on Fault-Tolerant Computing*, 97–104.

LIN, T.-H. AND SHIN, K. G. 1998. Damage assessment for optimal rollback-recovery. *IEEE Transactions on Computers 47*, 5, 603–613.

LITZKOW, M. AND SOLOMON, M. 1992. Supporting checkpointing and process migration outside the UNIX kernel. In *Usenix Winter 1992 Technical Conference*, 283–290.

LONG, J., FUCHS, W. K., AND ABRAHAM, J. A. 1990. Forward recovery using checkpointing in parallel systems. In *Proceedings of the 19th International Conference on Parallel Processing*, 272–275.

LONG, J., FUCHS, W. K., AND ABRAHAM, J. A. 1991. Implementing forward recovery using checkpointing in distributed systems. In *Proceedings of the International Conference on Dependable*

Computing for Critical Applications (DCCA), 20–27.

LONG, J., FUCHS, W. K., AND ABRAHAM, J. A. 1992. Compiler-assisted static checkpoint insertion. In *Digest of Papers, FTCS-22, The Twenty Second Annual International Symposium on Fault-Tolerant Computing*, 58–65.

LOWRY, A., RUSSELL, J. R., AND GOLDBERG, A. P. 1991. Optimistic failure recovery for very large networks. In *Proceedings, Tenth Symposium on Reliable Distributed Systems*, 66–75.

MANDELBERG, K. I. AND SUNDERAM, V. S. 1988. Process migration in UNIX networks. In *Proceedings of the Usenix Winter Technical Conference*, 357–364.

MANIVANNAN, D. AND SINGHAL, M. 1996. A low-overhead recovery technique using synchronous checkpointing. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, 100–107.

MANIVANNAN, D., NETZER, R. H., AND SINGHAL, M. 1997. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel & Distributed Systems 8*, 6, 623–627.

MATTERN, F. 1988. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, 215–226.

MCDERMID, J. A. 1982. Checkpointing and error recovery in distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, 271–282.

MERLIN, P. M. AND RANDELL, B. 1978. State restoration in distributed systems. In *Digest of Papers, FTCS-8, The Eighth Annual International Symposium on Fault-Tolerant Computing*, 129–134.

MITCHELL, J. R. AND GARG, V. K. 1998. A nonblocking recovery algorithm for causal message logging. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 3–9.

MOSTEFAOUI, A. AND RAYNAL, M. 1996. Efficient message logging for uncoordinated checkpointing protocols. In *Dependable Computing-EDCC-2, the Second European Dependable Computing Conference Proceedings*, 353–364.

MULLER, G., BANÂTRE, M., PEYROUZ, N., AND ROCHAT, B. 1996. Lessons from FTM: an experiment in design and implementation of a low-cost fault-tolerant system. *IEEE Transactions on Reliability 45*, 2, 332–340.

NETT, E., KROGER, R., AND KAISER, J. 1986. Implementing a general error recovery mechanism in a distributed operating system. In *Digest of Papers, FTCS-16, The Sixteenth Annual International Symposium on Fault-Tolerant Computing*, 124–129.

NETZER, R. B. AND MILLER, B. P. 1992. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing'92*, 502–511.

NETZER, R. B. AND XU, J. 1993. Adaptive message logging for incremental program replay. *IEEE Parallel and Distributed Technology 1*, 4, 32–39.

NETZER, R. B. AND WEAVER, M. H. 1994. Optimal tracing and incremental reexecution for Debugging Long-Running Programs. In *SIGPLAN '94: Conference on Programming Language Design and Implementation (PLDI)*, 313–325.

NETZER, R. B. AND XU, J. 1997. Replaying distributed programs without message logging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 137–147.

NEVES, N. AND FUCHS, W. K. 1996. Using time to improve the performance of coordinated checkpointing. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium, IPDS'96*, 282–291.

NEVES, N. AND FUCHS, W. K. 1997. Adaptive recovery for mobile environments. *Communications of ACM 40*, 1, 68–74.

NEVES, N. AND FUCHS, W. K. 1998. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *Digest of Papers, FTCS-28, The Twenty Eighth Annual International Symposium on Fault-Tolerant Computing*.

NEVES, N. AND FUCHS, W. K. 1998. Coordinated checkpointing without direct coordination. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS'98)*, 23–31.

NEVES, N., CASTRO, M., AND GUEDES, P. 1994. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the 1994 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*.

NICOLA, V. 1995. Checkpointing and the modeling of program execution time. In Lyu, M. ed. *Software Fault Tolerance*.

PARK, T. AND YEOM, H. Y. 2000. An asynchronous recovery scheme based on optimistic message logging for mobile computing systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS-20)*, 436–443.

PETERSON, S. L. AND KEARNS, P. 1993. Rollback based on vector time. In *Proceedings, Twelfth Symposium on Reliable Distributed Systems*, 68–77.

PETERSON, L. L., BUCHHOLZ, N. C., AND SCHLICHTING, R. D. 1989. Preserving and using context information in interprocess communication. *ACM Transaction on Computing Systems 7*, 3, 217–246.

PLANK, J. S. 1996. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *Proceedings, Fifteenth Symposium on Reliable Distributed Systems*, 76–85.

PLANK, J. S. AND ELWASIF, W. R. 1998. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Digest of Papers, FTCS-28, The Twenty Eighth Annual International Symposium on Fault-Tolerant Computing*, 48–57.

PLANK, J. S., BECK, M., AND KINGSLEY, G. 1995. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems Newsletter*, 62–67.

PLANK, J. S., KIM, Y., AND DONGARRA, J. J. 1995. Algorithm-based diskless checkpointing for fault-tolerant matrix computations. In *Digest of Papers, FTCS-25, The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, 351–360.

PLANK, J. S., YOUNGBAE, K., AND DONGARA, J. J. 1997. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel & Distributed Computing 43*, 2, 125–138.

PLANK, J. S., LI, K., AND PUENING, M. A. 1998. Diskless checkpointing. *IEEE Transactions on Parallel & Distributed Systems 9*, 10, 972–986.

PLANK, J. S., CHEN, Y., LI, K., BECK, M., AND KINGSLEY, G. 1996. *Memory exclusion: Optimizing the performance of checkpointing systems*. Technical Report UT-CS-96-335, University of Tennessee at Knoxville, Department of Computer Science.

POWELL, M. AND PRESOTTO, D. 1993. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, 100–109.

PRADHAN, D. K. AND VAIDYA, N. 1992. Roll-forward checkpointing scheme: Concurrent retry with non-dedicated spares. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 166–174.

PRADHAN, D. K. AND VAIDYA, N. 1994. Roll-forward and rollback-recovery: Performance-reliability trade-off. In *Digest of Papers, FTCS-24, The Twenty Fourth Annual International Symposium on Fault-Tolerant Computing*, 186–195.

RAMAMURTHY, B., UPADHYAYA, S., AND IYER, R. K. 1997. An object-oriented testbed for the evaluation of checkpointing and recovery systems. In *Digest of Papers, FTCS-27, The Twenty Seventh Annual International Symposium on Fault-Tolerant Computing*, 194–203.

RAMAMURTHY, B., UPADHYAYA, S., AND BHARGAVA, J. 1998. Design and analysis of a hardware-assisted checkpointing and recovery scheme for distributed applications. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 84–90.

RAMANATHAN, P. AND SHIN, K. G. 1988. Checkpointing and rollback-recovery in a distributed system using common time base. In *Proceedings, Seventh Symposium on Reliable Distributed Systems (SRDS-7)*, 13–21.

RAMANATHAN, P. AND SHIN, K. G. 1993. Use of common time base for checkpointing and rollback-recovery in a distributed system. *IEEE Transactions on Software Engineering SE-19*, 6, 571–583.

RAMKUMAR, B. AND STRUMPEN, V. 1997. Portable checkpointing for heterogeneous architectures. In *Digest of Papers, FTCS-22, The Twenty Second Annual International Symposium on Fault-Tolerant Computing*, 58–67.

RANGARAJAN, S., GARG, S., AND HUANG, Y. 1998. Checkpoints-on-demand with active replication. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 75–83.

RAO, S., ALVISI, L., AND VIN, H. 1999. Egida: An extensible toolkit for low-overhead fault tolerance. In *Digest of Papers, FTCS-29, The Twenty Ninth Annual International Symposium on Fault-Tolerant Computing*.

RUSSINOVICH, M. AND COGSWELL, B. 1996. Replay for concurrent non-deterministic shared memory applications. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 258–266.

RUSSINOVICH, M., SEGALL, Z., AND SIEWIOREK, D. P. 1993. Application transparent fault management in fault-tolerant Mach. In *Digest of Papers, FTCS-23, The Twenty Third Annual International Symposium on Fault-Tolerant Computing*, 10–19.

SCHWARZ, R. AND MATTERN, F. 1994. Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distributed Computing 7*, 149–174.

SELIGMAN, E. AND BEGUELIN, A. 1994. *High-level fault tolerance in distributed programs*. Technical Report CMU-CS-94-223, Carnegie Mellon University, School of Computer Science.

SHARMA, D. D. AND PRADHAN, D. K. 1994. An efficient coordinated checkpointing scheme for multicomputers. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*.

SILVA, L. M. AND SILVA, J. G. 1992. Global checkpointing for distributed programs. In *Proceedings, Eleventh Symposium on Reliable Distributed Systems*, 155–162.

SILVA, L. M. AND SILVA, J. G. 1994. Integrating a checkpointing and rollback-recovery algorithm with a causal order protocol. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, 523–540.

SILVA, L. M. AND SILVA, J. G. 1994. Checkpointing pipeline applications. In *Proceedings of the 1994 World Transputer Congress*, 497–512.

SILVA, L. M. AND SILVA, J. G. 1994. On the optimum recovery of distributed programs. In *Proceedings of the 20th EUROMICRO Conference*, 704–711.

SILVA, L. M. AND SILVA, J. G. 1996. A checkpointing facility for a heterogeneous DSM system. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 554–559.

SILVA, L. M. AND SILVA, J. G. 1998. Avoiding checkpoint contamination in parallel system. In *Digest of Papers, FTCS-28, The Twenty Eighth Annual International Symposium on Fault-Tolerant Computing*, 364–369.

SILVA, L. M. AND SILVA, J. G. 1998. An experimental study about diskless checkpointing. In *Proceedings of the 24th EUROMICRO Conference*, 395–402.

SILVA, L. M. AND SILVA, J. G. 1998. System-level versus user-defined checkpointing. In *Proceedings, Seventeenth Symposium on Reliable Distributed Systems*, 68–74.

SILVA, L. M., VEER, B., AND SILVA, J. G. 1994. Checkpointing SPMD applications on transputer networks. In *Proceedings of the Scalable High-Performance Computing Conference, SHPCC94*, 694–701.

SILVA, L. M., SILVA, J. G., AND CHAPPLE, S. 1996. Portable transparent checkpointing for distributed shared memory. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, HPDC-5*, 422–431.

SILVA, L. M., TAVORA, V. N., AND SILVA, J. G. 1996. Mechanisms of file-checkpointing for UNIX applications. In *Proceedings of the 14th IASTED Conference on Applied Informatics*, 358–361.

SILVA, L. M., SILVA, J. G., CHAPPLE, S., AND CLARKE, L. 1995. Portable checkpointing and recovery. In *Proceedings of the 4th International Symposium on High-Performance Distributed Computing, HPDC-4*, 188–195.

SINHA, A., DAS, P. K., AND CHAUDHURI, A. 1992. Checkpointing and recovery in a pipeline of transputers. In *Proceedings of Euromicro'92*, 141–148.

SLYE, J. H. 1996. *Adding support for software interrupts in log-based rollback-recovery protocols*. Master Thesis, Carnegie Mellon University, Department of Computer Science.

SLYE, J. H. AND ELNOZAHY, E. N. 1996. Supporting nondeterministic execution in fault-tolerant systems. In *Digest of Papers, FTCS-26, The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*.

SMITH, J. M. AND IOANNIDIS, J. 1989. Implementing remote fork() with checkpoint/restart. *IEEE Technical Committee on Operating Systems Newsletter*, 12–16.

SOLIMAN, H. M. AND ELMAGHRABY, A. S. 1998. An analytical model for hybrid checkpointing in time warp distributed simulation. *IEEE Transactions on Parallel & Distributed Systems 9*, 10, 947–951.

SPEZIALETTI, M. AND KEARNS, P. 1986. Efficient distributed snapshots. In *Proceedings of the International Conference on Distributed Computing Systems*, 382–388.

SSU, K.-F. AND FUCHS, W. K. 1998. PREACHES-portable recovery and checkpointing in heterogeneous systems. In *Digest of Papers, FTCS-28, The Twenty Eighth Annual International Symposium on Fault-Tolerant Computing*, 38–47.

STAINOV, R. 1991. An asynchronous checkpointing service. *Microprocessing and Microprogramming 31*, 117–120.

STAKNIS, M. 1989. Sheaved memory: Architectural support for state saving and restoration in paged systems. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, 96–102.

STELLNER, G. 1994. Consistent checkpoints of PVM applications. In *Proceedings of the First European PVM User Group Meeting*.

STELLNER, G. 1996. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*.

STROM, R. E., BACON, D. F., AND YEMINI, S. A. 1988. Volatile logging in n-fault-tolerant distributed systems. In *Digest of Papers, FTCS-18, The Eighteenth Annual International Symposium on Fault-Tolerant Computing*, 44–49.

STROM, R. E., YEMINI, S. A., AND BACON, D. F. 1988. A recoverable object store. In *Proceedings of the Hawaii International Conference on System Sciences*, II-215–II221.

SURI, G., JANSSENS, B., AND FUCHS, W. K. 1995. Reduced overhead logging for rollback-recovery in distributed shared memory. In *Digest of Papers, FTCS-25, The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, 279–288.

SURI, G., HUANG, Y., WANG, Y. M., FUCHS, W. K., AND KINTALA, C. 1995. An implementation and performance measurement of the progressive retry technique. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, 41–48.

TAM, V.-O. AND HSU, M. 1990. Fast recovery in distributed shared virtual memory systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, 38–45.

TAMIR, Y. AND GAFNI, E. 1987. A software-based hardware fault tolerance scheme for multicomputers. In *Proceedings of the International Conference on Parallel Processing*, 117–120.

TAMIR, Y. AND FRAZIER, T. M. 1989. Application-transparent process-level error recovery for multicomputers. In *Proceedings of the Hawaii International Conferences on System Sciences-22*, 296–305.

TAMIR, Y. AND FRAZIER, T. M. 1991. *Error-recovery in multicomputers using asynchronous coordinated checkpointing*. Technical Report CSD-010066, University of California.

TANAKA, K. AND TAKIZAWA, M. 1996. Distributed checkpointing based on influential messages. In *Proceedings of the 1996 International Conference on Parallel and Distributed Systems*, 440–447.

TANAKA, K., HIGAKI, H., AND TAKIZAWA, M. 1998. Object-based checkpoints in distributed systems. *Computer Systems Science & Engineering 13*, 3, 179–185.

TAYLOR, D. J. AND WRIGHT, M. L. 1986. Backward error recovery in a UNIX environment. In *Digest of Papers, FTCS-16, The Sixteenth Annual International Symposium on Fault-Tolerant Computing*, 118–123.

THANAWASTIAN, S., PAMULA, R. S., AND VAROL, Y. L. 1986. Evaluation of global checkpoint rollback strategies for error recovery in concurrent processing systems. In *Digest of Papers, FTCS-16, The Sixteenth Annual International Symposium on Fault-Tolerant Computing*, 246–251.

TONG, Z., KAIN, R. Y., AND TSAI, W. T. 1989. A lower overhead checkpointing and rollback-recovery scheme for distributed systems. In *Proceedings, Eighth Symposium on Reliable Distributed Systems*, 12–20.

TSAI, J., KUO, S. Y., AND WANG, Y.-M. 1998. Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability. *IEEE Transactions on Parallel & Distributed Systems 9*, 10, 963–971.

TSURUOKA, K., KANEKO, A., AND NISHIHARA, Y. 1981. Dynamic recovery schemes for distributed processes. In *Proceedings of the IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, 124–130.

TULLMANN, P., LEPREAU, J., FORD, B., AND HIBLER, M. 1996. User-level checkpointing through exportable kernel state. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems*, 85–88.

VAIDYA, N. H. 1993. *Dynamic cluster-based recovery: Pessimistic and optimistic schemes*. Technical Report 93-027, Texas A&M University, Department of Computer Science.

VAIDYA, N. H. 1995. A case of two-level distributed recovery schemes. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'95)*, 64–73.

VAIDYA, N. H. 1996. On staggered checkpointing. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, 572–580.

VENKATESAN, S. 1989. Message-optimal incremental snapshots. In *Proceedings of the International Conference on Distributed Computing Systems*, 53–60.

VENKATESAN, S. 1997. Optimistic crash recovery without changing application messages. *IEEE Transactions on Parallel and Distributed Systems 8*, 3, 263–271.

VENKATESH, K., RADAKRISHNAN, T., AND LI, H. L. 1987. Optimal checkpointing and local recording for domino-free rollback-recovery. *Information Processing Letters 25*, 295–303.

WANG, Y. M. 1993. Reducing message logging overhead for log-based recovery. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1925–1928.

WANG, Y. M. 1995. The maximum and minimum consistent global checkpoints and their applications. In *Proceedings, Fourteenth Symposium on Reliable Distributed Systems*.

WANG, Y.-M. AND FUCHS, W. K. 1992. Optimistic message logging for independent checkpointing in message passing systems. In *Proceedings, Eleventh Symposium on Reliable Distributed Systems*, 147–154.

WANG, Y. M. AND FUCHS, K. 1992. Scheduling message processing for reducing rollback propagation. In *Digest of Papers, FTCS-22, The Twenty Second Annual International Symposium on Fault-Tolerant Computing*, 204–211.

WANG, Y. M. AND FUCHS, K. 1993. Lazy checkpoint coordination for bounding rollback propagation. In *Proceedings, Twelfth Symposium on Reliable Distributed Systems*, 78–85.

WANG, Y. M. AND FUCHS, W. K. 1994. Optimal message log reclamation for uncoordinated checkpointing. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*.

WANG, Y. M., HUANG, Y., AND FUCHS, W. K. 1993. Progressive retry for software error recovery in distributed systems. In *Digest of Papers, FTCS-23, The Twenty Third Annual International Symposium on Fault-Tolerant Computing Systems*, 138–144.

WANG, Y. M., LOWRY, A., AND FUCHS, W. K. 1994. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters 50*, 4, 223–230.

WANG, Y. M., HUANG, Y., AND KINTALA, C. 1997. Progressive retry for software failure recovery in message passing applications. *IEEE Transactions on Computers 46*, 10, 1137–1141.

WANG, Y. M., DAMANI, O. P., AND GARG, V. K. 1997. Distributed recovery with K-optimistic logging. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, 60–67.

WANG, Y. M., CHUNG, E., HUANG, Y., AND ELNOZAHY, E. N. 1997. Integrating checkpointing with transaction processing. In *Digest of Papers, FTCS-27, The Twenty Seventh Annual International Symposium on Fault-Tolerant Computing*, 304–308.

WANG, Y. M., HUANG, Y., VO, K. P., CHUNG, P. Y., AND KINTALA, C. 1995. Checkpointing and its applications. In *Digest of Papers, FTCS-25, The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, 22–31.

WEI, X. AND JU, J. 1998. A consistent checkpointing algorithm with shorter freezing time. *Operating Systems Reviews 32*, 4, 70–76.

WOJCIK, Z. AND WOJCIK, B. E. 1990. Fault-tolerant distributed computing using atomic send receive checkpoints. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, 215–222.

WONG, K. F. AND FRANKLIN, M. 1996. Checkpointing in distributed computing systems. *Journal of Parallel & Distributed Computing*, 67–75.

WOOD, W. G. 1981. A decentralized recovery control protocol. In *Digest of Papers, FTCS-11, The Eleventh Annual International Symposium on Fault-Tolerant Computing*, 159–164.

WOOD, W. G. 1995. Recovery control of communicating processes in a distributed system. In Shrivastava, S. K. ed. *Reliable Computing Systems*, Springer Verlag.

WU, K. L. AND FUCHS, W. K. 1990. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers 39*, 4, 460–469.

WYNER, D. S. 1972. A technique for optimizing the performance of a checkpoint restart system. In *Proceedings of the Canadian Computer Conference* (Montreal), 201–212.

XU, J. AND NETZER, R. H. B. 1993. Adaptive independent checkpointing for reducing rollback propagation. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, 754–761.

XU, J., NETZER, R. B., AND MACKEY, M. 1995. Sender-based message logging for reducing rollback propagation. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, 602–609.

YOUNG, J. W. 1974. A first order approximation to the optimum checkpoint interval. *Communications of the ACM 17*, 9.

ZAMBONELLI, F. 1998. Distributed checkpoint algorithms to avoid rollback propagation. In *Proceedings of the 24th EUROMICRO Conference*, 403–410.

ZIV, A. AND BRUCK, J. 1994. Efficient checkpointing over local area networks. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 30–35.

ZIV, A. AND BRUCK, J. 1997. An on-line algorithm for checkpoint placement. *IEEE Transactions on Computers 46*, 9, 976–985.

ZWEIACKER, M. 1997. Fault-tolerant CORBA using checkpointing and recovery. *Comtec 75*, 8, 20–25.