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

Mootaz Elnozahy[*] Lorenzo Alvisi[†]
Yi-Min Wang[‡] David B. Johnson[§]

June 1999
CMU-CS-99-148
(A revision of CMU-CS-96-181)

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Mootaz Elnozahy is with IBM Austin Research Lab. Email: mootaz@us.ibm.com.
[†]Lorenzo Alvisi is with the Department of Computer Sciences, University of Texas at Austin. Email: lorenzo@cs.utexas.edu.
[‡]Yi-Min Wang is with Microsoft Research. Email: ymwang@microsoft.com.
[§]David B. Johnson is with the Computer Science Department, Carnegie Mellon University. Email: dbj@cs.cmu.edu.

Keywords: Distributed systems, fault tolerance, high availability, checkpointing, message logging, rollback, recovery.

Abstract

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.

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 communications and rollback recovery, and have different tradeoffs and focuses. For example, transactions focus on data-oriented applications, while group communications offer an abstraction of an ideal communication system on top of which programmers can develop reliable applications. This survey covers transparent rollback recovery, which focuses on long-running applications such as scientific computing and telecommunication applications [26][43].

Rollback recovery treats a distributed system as a collection of application processes that communicate through a network. Fault tolerance is achieved by periodically using stable storage to save the processes' states during failure-free execution. Upon a failure, a failed process restarts from one of its saved states, thereby reducing the amount of lost computation. Each of the saved states is called a *checkpoint*.

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 [47]. 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 hardware-level instruction retry, distributed shared memory [38], real-time systems, and debugging [37]. 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 [51]. Also excluded are rollback-recovery techniques that rely on special language constructs such as recovery blocks [47] 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* [47].

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 [16]. 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 it receives from other processes [50]. 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 [56], 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* [1]. 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.

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 [16], while other protocols assume that the communication subsystem can lose, duplicate, or reorder messages [28]. 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 Section 2.4 and Section 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 process may fail, in which case it loses its volatile state and stops execution according to the fail-stop model [51]. 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.

[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.

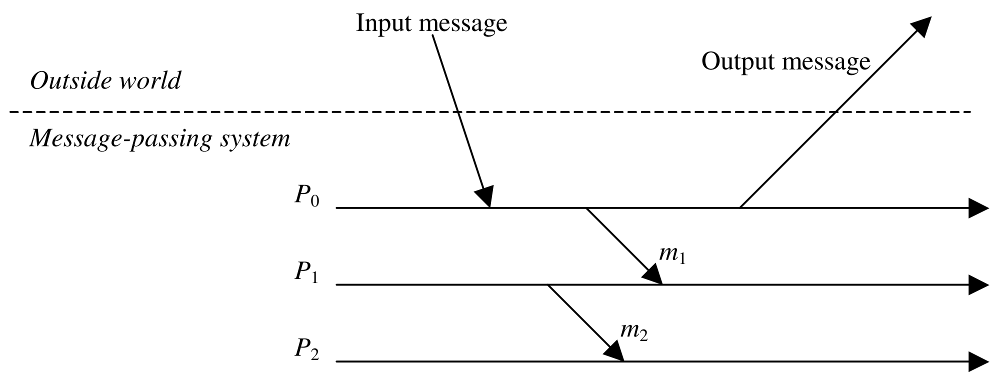

Figure 1. An example of a message-passing system with three processes.

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 a process's state reflects a message receipt, then the state of the corresponding sender reflects sending that message [16]. For example, Figure 2 shows two examples of global states—a consistent state in Figure 2(a), and an inconsistent 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.

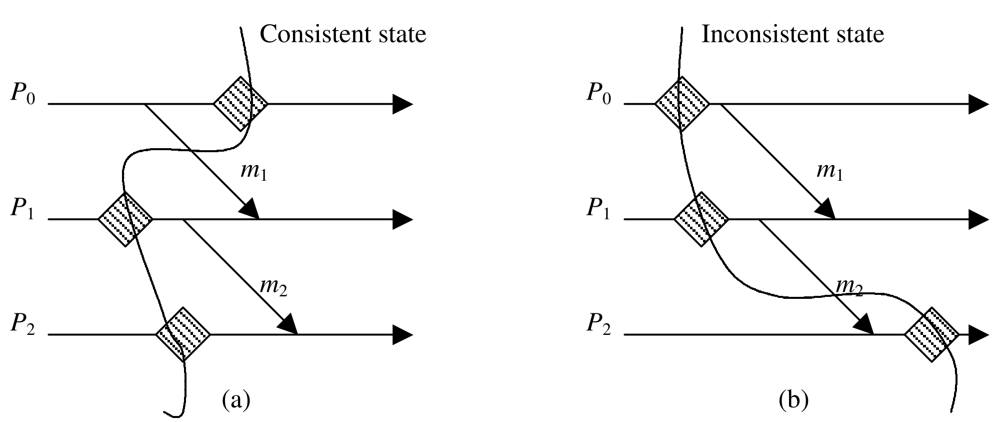

Figure 2. An example of a consistent and inconsistent state.

2.3 Z-Paths and Z-Cycles

A *Z-path* (zigzag path) is a special sequence of messages that connects two checkpoints [41]. Let $\mapsto$ denote Lamport's happen-before relation [34]. 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 $[m_0, m_1, \ldots, m_n]$, $n \geq 0$, 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 3, $[m_1, m_2]$ 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 3, the Z-path $[m_5, m_3, m_4]$ is a Z-cycle that starts and ends at checkpoint $c_{2,2}$.

The Z-cycle theory was first introduced as a framework for reasoning about consistent system states. Recently, the theory has proved a powerful tool for reasoning about a class of protocols known as communication-induced checkpointing [5][24]. In particular, it has been proven that a checkpoint involved in a Z-cycle cannot become part of a consistent state in a system that uses only checkpoints.

2.4 In-Transit Messages

In Figure 2(a), the global state shows that message m_1 has been sent but not yet received. We call such a message an *in-transit* message. When in-transit messages are part of a global system state, these messages 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 4(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 4(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

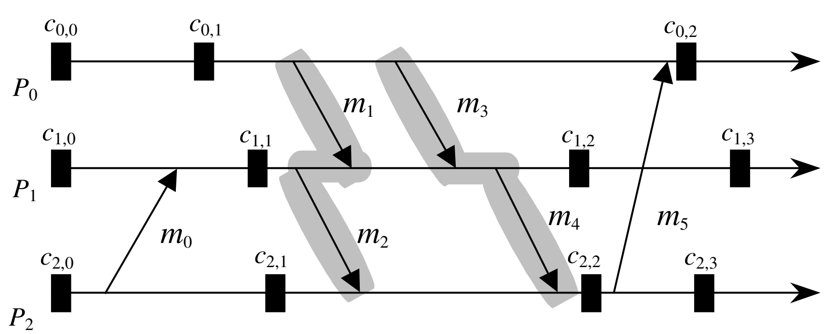

Figure 3. An example execution and Z-paths.

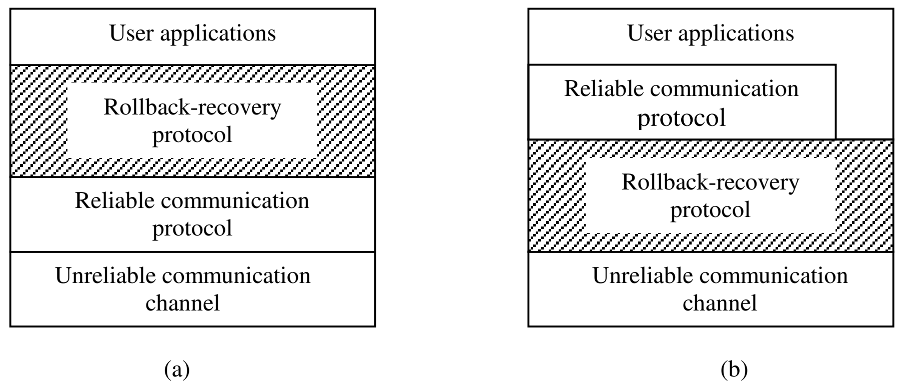

Figure 4. Implementation of rollback-recovery (a) on top of a reliable communication protocol; (b) directly on top of unreliable communication channels.

model assumes unreliable communication channels, as in Figure 4(b), then the recovery protocol need not handle in-transit messages in any special way. Indeed, lost in-transit messages because of process failures cannot be distinguished from those caused by 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 Checkpointing Protocols and the Domino Effect

In checkpointing protocols, each process periodically saves its state on stable storage. The saved state contains sufficient information to restart process execution. A *consistent global checkpoint* is a set of N local checkpoints, one from each process, forming a consistent system state. Any consistent global checkpoint can be used to restart process execution upon a failure. Nevertheless, it is desirable to minimize the amount of lost work by restoring the system to the most recent consistent global checkpoint, which is called the *recovery line* [47].

Processes may coordinate their checkpoints to form consistent states, or may take checkpoints independently and search for a consistent state during recovery out of the set of saved individual checkpoints. The second style, however, can lead to the *domino effect* [47]. For example, Figure 5 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.

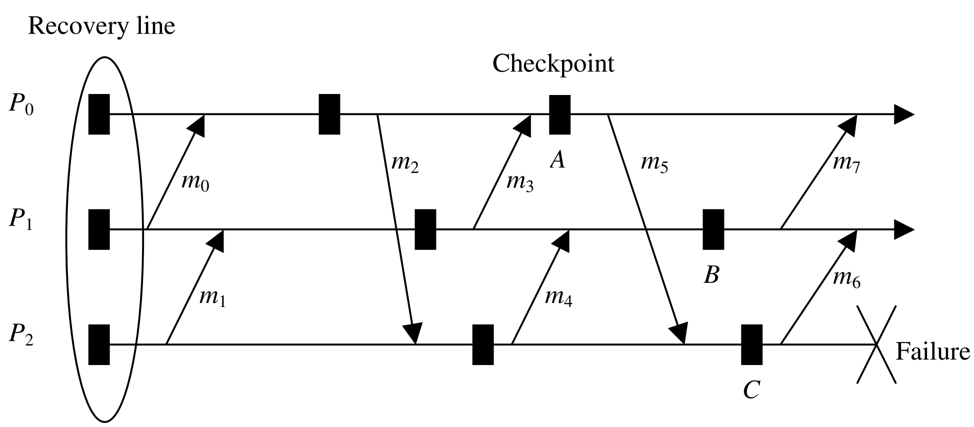

Figure 5. Rollback propagation, recovery line and the domino effect.

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 5, cascading rollbacks due to the single failure of process P_2 may result in a recovery line that consists of the initial set of checkpoints, effectively causing the loss of all the work done by all processes. To avoid the domino effect, processes need either to coordinate their checkpoints so that the recovery line is advanced as new checkpoints are taken, or to combine checkpointing with event logging.

2.6 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 [42]. 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. It is therefore necessary that the outside world perceive a consistent behavior of the system despite failures. Thus, before sending output to the outside world, the system must ensure that the state from which the output is sent will be recovered despite any future failure. This is commonly called the *output commit* problem [56]. Similarly, input messages that a system receives from the outside world may not be reproducible during recovery, because it may not be possible for the outside world 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 the outside world after it has been issued by the system, or the time it takes a process to consume an input message after it has been posted to the system. The second metric represents the overhead that the system incurs to ensure that input and output actions will have a persistent effect despite future failures.

2.7 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 output sent to the outside world without having to take expensive checkpoints before sending such output. 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 [56]. 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 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 [30]. A recoverable state interval is always stable, but the opposite is not always true [28].

Figure 6 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

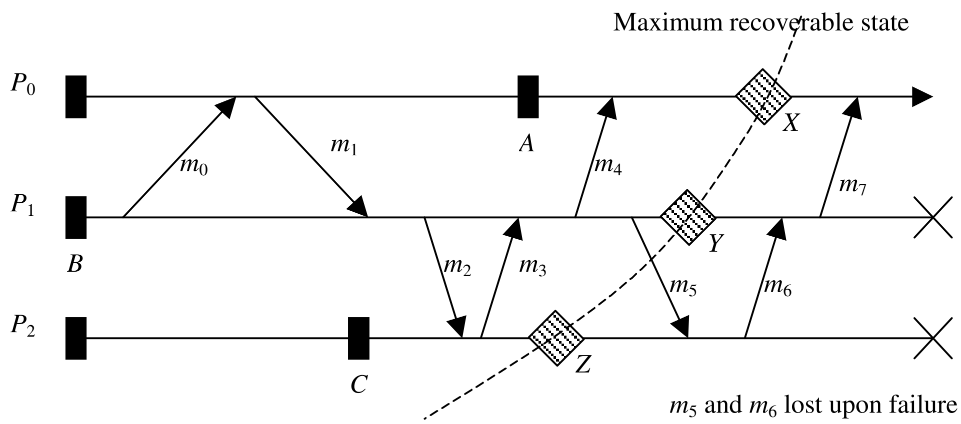

Figure 6. Message logging for deterministic replay.

as well. States *X*, *Y* and *Z* form the *maximum recoverable state* [28], i.e., 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.

2.8 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 [11][29].
- 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 [35].

2.9 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 recovery line 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, i.e. the recovery line [47]. 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 during recovery be the same as during failure-free operation.

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 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 [59]. 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 invoke periodically 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.

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 [9]. Let $c_{i,x}$ be the x^{th} 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 7, 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 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, which 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.

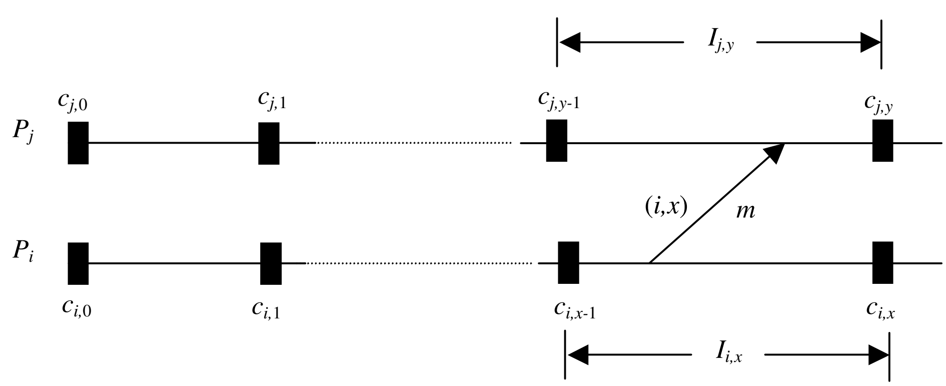

Figure 7. Checkpoint index and checkpoint interval.

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* [9] 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 8(b) shows the rollback dependency graph for the execution in Figure 8(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 [9] 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 8(b).

The second approach is based on the *checkpoint graph* [59]. 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 8(c). The recovery line can be calculated by first removing both the nodes 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* [59] on the checkpoint graph as shown in Figure 9. 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).

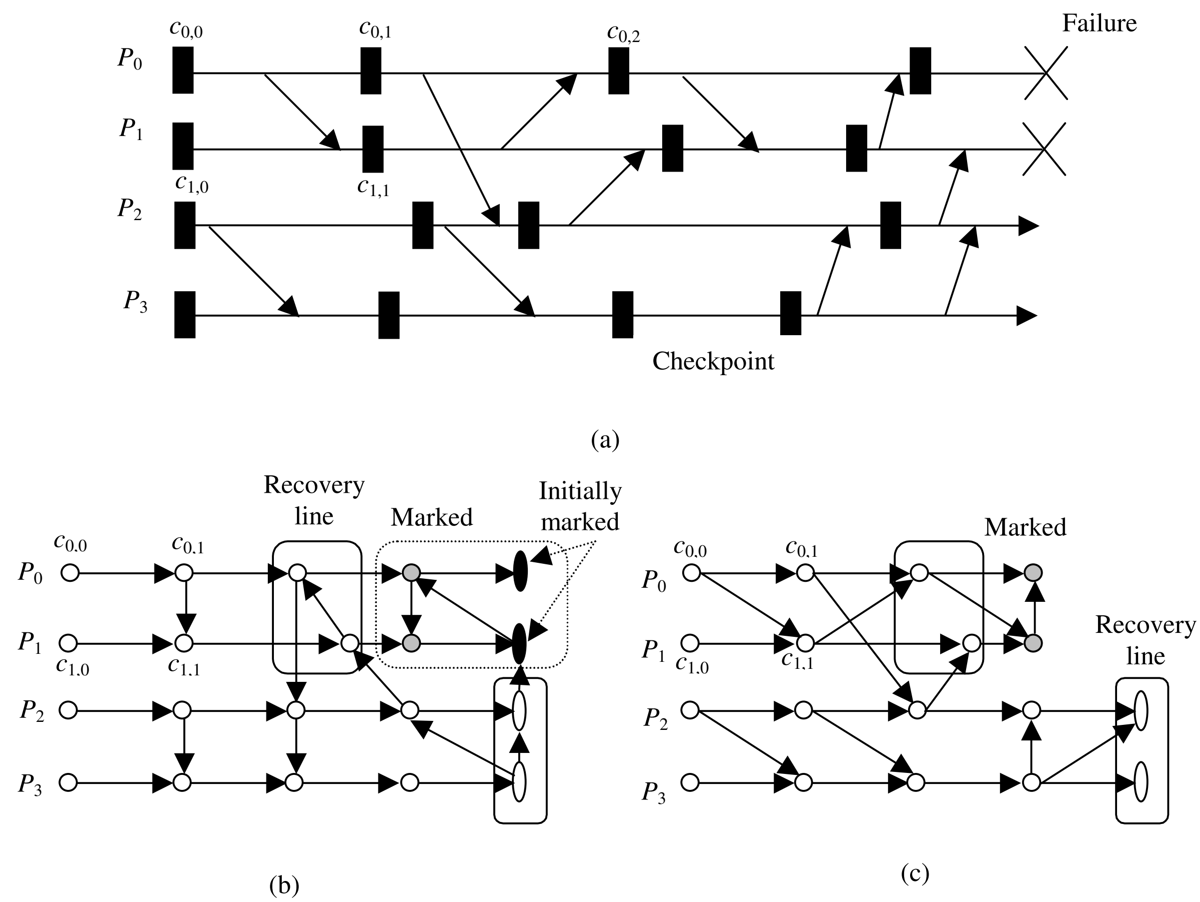

Figure 8. (a) Example execution; (b) rollback-dependency graph; (c) checkpoint graph.

3.1.3 Garbage Collection

Any checkpoint that precedes the recovery lines for all possible combinations of process failures can be garbage-collected. The garbage collection algorithm based on a rollback dependency graph works by identifying the obsolete checkpoints as follows. First, it marks all volatile checkpoints and removes all edges ending in a marked checkpoint, producing a *non-volatile rollback dependency graph* [63]. Then, it uses reachability analysis to determine the worst-case recovery line for this graph, called the global recovery line. Figure 10 shows the non-volatile rollback-dependency graph and the global recovery line of Figure 8(a). In this case, only the first checkpoint of each process is obsolete and can be garbage-collected. As the figure illustrates, when the global recovery line is unable to advance because of rollback propagation, a large number of non-obsolete checkpoints may need to be retained.

By deriving the necessary and sufficient conditions for a checkpoint to be useful for any future recovery, it is possible to derive an optimal garbage collection algorithm that reduces the number of retained checkpoints [62]. The algorithm determines a set of N recovery lines, the union of which contains all useful checkpoints. Each of the N recovery lines is obtained by initially marking one volatile checkpoint in the non-volatile rollback-dependency graph. For the graph in Figure 10, the optimal algorithm identifies the four checkpoints A,B,C and D as well as the four obsolete checkpoints as garbage checkpoints. The number of useful checkpoints has a tight upper bound of $N(N+1)/2$ [62].

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 output can be committed to the outside world.

A straightforward approach to coordinated checkpointing is to block communications while the checkpointing protocol executes [57]. 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 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 [20].

```
include last checkpoint of each failed process as an element in set RootSet;
include current state of each surviving process as an element in RootSet;
mark all checkpoints reachable by following at least one edge from any member of RootSet;
while (at least one member of RootSet is marked)
        replace each marked element in RootSet by the last unmarked checkpoint of the
                same process;
        mark all checkpoints reachable by following at least one edge from any member
                of RootSet
endwhile
RootSet is the recovery line.
```

Figure 9. The rollback propagation algorithm.

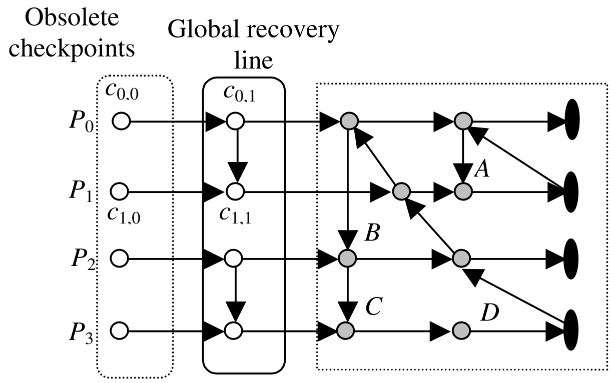

Figure 10. Garbage collection based on global recovery line and obsolete checkpoints.

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 11(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 11(b). An example of a non-blocking checkpoint coordination protocol using this idea is the *distributed snapshot* [16], 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 11(c) [33]. 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 [20][52].

3.2.3 Synchronized Checkpoint Clocks

Loosely synchronized clocks can facilitate checkpoint coordination [58]. More specifically, loosely synchronized clocks can trigger the local checkpointing actions of all participating processes at approximately the same time without a checkpoint initiator. 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

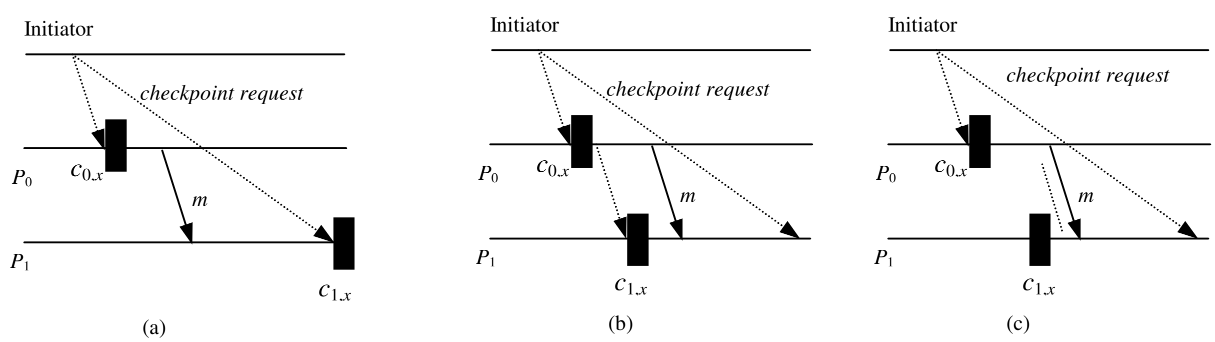

Figure 11. Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) with FIFO channels; (c) non-FIFO channels (the short dashed line represents a piggybacked *checkpoint request*).

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. To guarantee checkpoint consistency, either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking as explained above.

3.2.4 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 only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints [32].

The following two-phase protocol achieves minimal checkpoint coordination [32]. 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 allowable.

3.3 Communication-induced Checkpointing

3.3.1 Overview

Communication-induced checkpointing avoids the domino effect while allowing processes to take some of their checkpoints independently [14]. However, process independence is constrained to guarantee the eventual progress of the recovery line, and therefore processes may be forced to take additional checkpoints. The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints. Communication-induced checkpointing piggybacks protocol-related information on each application message. The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line. The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring high latency and overhead. It is therefore desirable in these systems to reduce the number of forced checkpoints to the extent possible. In contrast with coordinated checkpointing, no special coordination messages are exchanged.

We distinguish two types of communication-induced checkpointing. In *model-based checkpointing*, the system maintains checkpoint and communication structures that prevent the domino effect or achieve some even stronger properties [60]. In *index-based coordination*, the system uses an indexing scheme for the local and forced checkpoints, such that the checkpoints of the same index at all processes form a consistent state. Recently, it has been proved that the two types are fundamentally equivalent [25], although in practice, there may be some evidence that index-based coordination results in fewer forced checkpoints [2]. Other practical issues concerning these protocols will be discussed in Section 5.

3.3.2 Model-based Checkpointing

Model-based checkpointing relies on preventing patterns of communications and checkpoints that could result in inconsistent states among the existing 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. Model-based checkpointing works with the restriction that no control (out-of-band) messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggybacked on top of application messages. The decision to force a checkpoint is done locally using the information available. Therefore, under this style of checkpointing it is possible that two 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, this style of 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 [50] 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 [60]. Another way to prevent the domino effect is to avoid rollback propagation completely by taking a checkpoint immediately after every message-sending event [7]. Recent work has focused on ensuring that every checkpoint can belong to a consistent global checkpoint and therefore is not useless [5][24][25][41].

3.3.3 Index-based Communication-Induced Checkpointing

Index-based communication-induced checkpointing works by assigning monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state [14]. The indexes 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 [14]. More sophisticated protocols piggyback more information on top of application messages to minimize the number of forced checkpoints [24].

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 [56]. 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 5, 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 [7][11][56]. 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, i.e., 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 [34].
- *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.

Now, suppose that a set of processes ψ crashes. A process p in ψ 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 [1]:

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

We call this property the *always-no-orphans* condition. It stipulates that if any surviving process depends on an event e, that 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 12. 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 deliver again the same sequence of messages as in the pre-failure execution. This guarantees that P_1 and P_2 will repeat exactly 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 commit output 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 on-going 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 or with the outside world. This is highly desirable in practical systems [27].

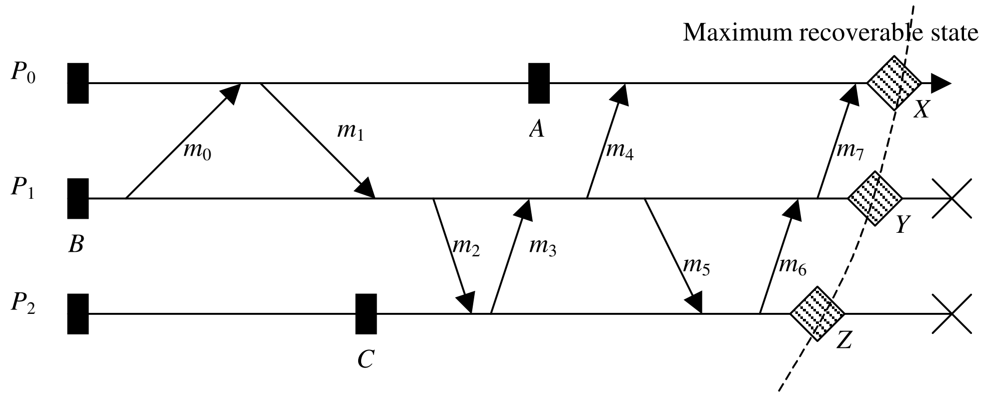

Figure 12. Pessimistic logging.

4. Recovery information can be garbage-collected easily. 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 [11], while others may limit the number of tolerated failures to improve performance [29][31].

4.2.2 Techniques for Reducing Performance Overhead

Synchronous logging [11] 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 [6]. 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 [11]. 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 [7][11].

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 [29]. The determinant of *m*, which consists of its content and the order in which it was 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 to the determinant the ordering information. 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 [31].

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 host communicates with another host or with the outside world [28][29]. In the example of Figure 12, 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 or the outside world. 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 12, 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 assumption about reliable channels. This problem does not arise in protocols that log messages at the sender or do not assume reliable communication channels [18][28][29][30].

4.3 Optimistic Logging

4.3.1 Overview

In optimistic logging protocols, processes log determinants *asynchronously* to stable storage [56]. 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, 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 13 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 [28][30][53], 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 [28][53], 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

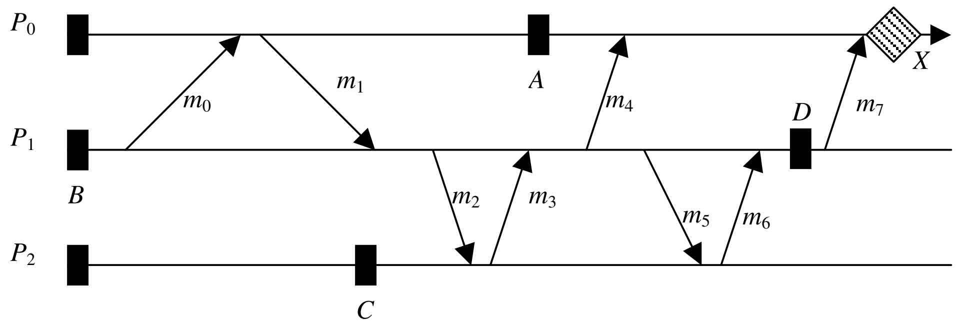

Figure 13. Optimistic logging.

dependency tracking [53][56] 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* [55][56]. 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 [55].

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 [53]. Figure 14 gives an example, where each integer pair $[i,x]$ represents the x^{th} state interval of the i^{th} 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. By distinguishing failure announcements from rollback announcements and broadcasting only the former, the source of the exponential-rollbacks problem is eliminated [53]. 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 14, process P_1 piggybacks r_0 on r_1. Exponential rollbacks can be avoided by piggybacking all rollback announcements on every application message [55].

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 [1][18]. 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 processes 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.

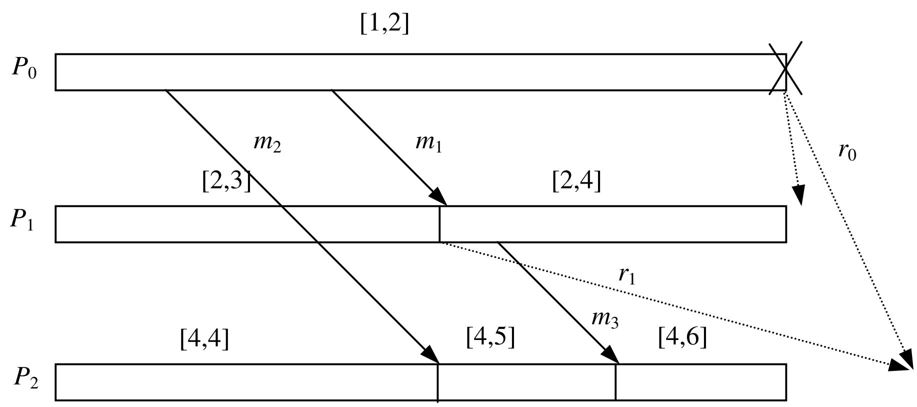

Figure 14. Exponential rollbacks.

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 15(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 [34]. 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 implements 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* [18]. 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 precedes the state of a process, and the edges correspond to the *happened-before* relation [34]. Figure 15(b) shows the antecedence graph of process P_0 of Figure 15(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, i.e., 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 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, and therefore transmitting them serves no purpose. Other optimizations are also possible but depend on the semantics of the communication protocol [18]. An implementation of this technique shows that it has very low overhead in practice [18].

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 [1]. 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 [1]. 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 properties such as 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 1 provides a brief comparison between the different styles 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 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 [62]. Also,

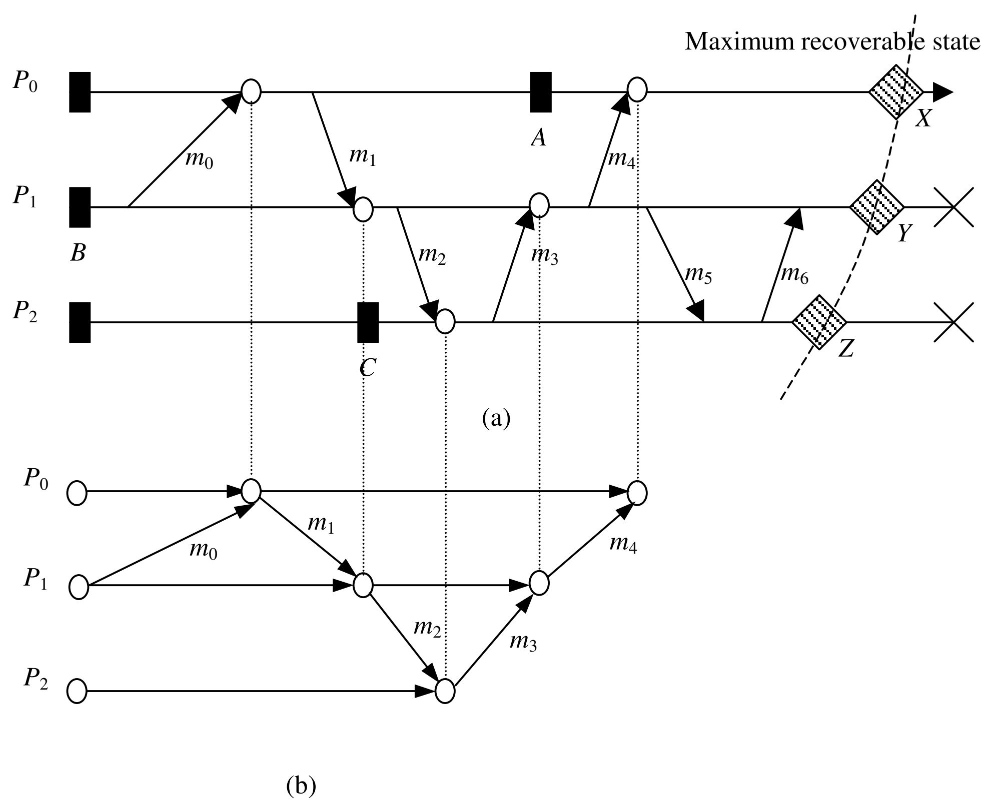

Figure 15. Causal logging. (a) Maximum recoverable states, and (b) antecedence graph of P_0 at state X.

	Uncoordinated Checkpointing	Coordinated Checkpointing	Communication Induced Checkpointing	Pessimistic Logging	Optimistic Logging	Causal Logging
PWD assumed?	No	No	No	Yes	Yes	Yes
Garbage collection	Complex	Simple	Complex	Simple	Complex	Complex
Checkpoints per 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
Complex recovery?	Yes	No	Yes	No	Yes	Yes
Output commit	Not possible	Very slow	Very slow	Fastest	Slow	Fast

Table 1. Comparison between different styles of rollback-recovery protocols.

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 [1][18][20][26][28][39][43][44][48][49][52].

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 [18]. It is interesting that all commercial implementations of message logging use pessimistic logging because it simplifies recovery [11][27].

Several recent studies have also challenged some premises on which many rollback-recovery protocols rely. Many of these protocols have been incepted in the 1980's, when processor speed and network bandwidth were such that communication overhead was deemed too high, especially when compared to the cost of stable storage access [10]. In such platforms, multi-host coordination incurs a large overhead because of the necessary control messages. A protocol that does not require such communication overhead at the expense of more stable storage access 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.[1] This change in the equation suggests a

[1] 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.

fresh look at the premises of many rollback-recovery protocols and recent results have shown that [1][18][28][39][43][52][54]:

- 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 [21]. 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 [43]. 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 [57]. 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 [23][43]. 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 [22] to concurrent checkpointing can further reduce the overhead [20]. 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 [4]. A public domain package implementing this technique along with concurrent checkpointing is available [44].

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 [45]. 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 [46]. 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 [40]. 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 is unsafe in practice [19].

5.2.3 System-level versus User-level Implementations

Support for checkpointing can be implemented in the kernel [7][11][18][28], or it can be implemented by a library linked with the user program [1][23][26][44]. 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 [23]. 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 [26].

5.2.4 Compiler Support

A compiler can be instrumented to generate code that supports checkpointing [36]. A compiled program would contain 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 [8][44]. The compiler may also be trained by running the application in an iterative manner and observing its behavior [36]. 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 [3]. 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 [15]. The problem is usually formulated as an optimization problem subject to constraints. For instance, a system may attempt to reduce the number of checkpoints taken 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, and therefore the optimality f checkpoint placement is rarely an issue in practical systems [20].

5.3 Checkpointing Protocols in Comparison

Many checkpointing protocols were incepted at a time where 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 checkpointing schemes. 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 [1][18][28][39][43][52]. 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 the balance.

A recent study has also shed some light on the behavior of communication-induced checkpointing [2]. 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 CIC 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 inter-process 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 redirect communication appropriately to a restarting process after a failure [18].

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 on 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 [21]. 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 [21].

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 [21]. 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 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 [21]. 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 [35]. Although they are slow, their storage capacity and low cost combination cannot be matched with 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 [6][18][49]. 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 [49]. The package runs in conventional UNIX systems and bypasses the UNIX file system by accessing the disk in raw mode. There have been also several attempts at implementing stable storage using non-volatile semiconductor memory [6]. Such implementations do not have the performance problems associated with disks. 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. The *PWD* assumption inherent in log-based recovery systems stipulates that these nondeterministic events 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 under the *PWD* assumption. There are two main sources of nondeterminism in actual systems, namely system calls and asynchronous events.

5.7.1 System Calls

System calls in general can be classified into three types [11][18][23]. 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 result 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 are 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, which can be very expensive [7]. Alternatively, the system may convert these asynchronous signals to synchronous messages such as in Targon/32 [11], 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 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 multi-threaded 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 [23]. 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* [13]. 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 [37].

Instruction counters can be used in rollback recovery to track the number of instructions that occur between asynchronous interrupts [54]. 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 [13]. It also can be emulated in software [37]. 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 [54].

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 messages [14], while some require the propagation of a vector of timestamps [56]. At an extreme, some protocols require the propagation of a graph describing the history of the computation [18], or matrices containing bit or timestamp vectors [5].

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* has 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 [18].

The other technique for reducing the overhead of dependency tracking exploits the semantics of the applications and the communication patterns [18]. 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 [18]. Regardless of the particular method used to track inter-process dependencies, various prototype implementations have shown that the overhead resulting from tracking is negligible compared to the overhead of checkpointing or logging [1][2][9][11][18][23][28][48][52].

5.9 Recovery

Handling execution restart and replay is a difficult part of implementing a rollback-recovery system [11]. 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 [18]. 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 [18][26][28][43]. 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 imitate 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 structure indirectly. Obviously, not all state within the operating system kernel can be emulated this way, and therefore, out-of-kernel implementations will have to have stricter coverage of the system's state that need to 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 [44].

5.9.2 Behavior During Recovery

Previous studies have outlined several characteristics of rollback-recovery systems during recovery [18][48]. 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 [56], 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 [48]. It is possible to address this tradeoff by performing logging both at the sender and receivers [56], such that the sender log is volatile and is kept only until the receiver flushes its volatile logs to stable storage.

5.10 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 them. 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 [44].

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 [11]. 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 [26], 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 not aware, 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 based on the *PWD* assumption is often a natural choice for applications that frequently interact with the outside world. Log-based recovery allows efficient output commit, and has three flavors, pessimistic, optimistic, and causal. The simplicity of pessimistic logging makes it attractive for practical applications if a high failure-free overhead can be tolerated. This form of logging simplifies recovery, output commit, and protect surviving processes from having to roll back. These advantages have made pessimistic logging attractive in commercial environment where simplicity and robustness are necessary. Causal logging can be employed to reduce 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, and Paulo Verissimo for valuable discussions, encouragement, and comments.

References

[1] L. Alvisi. "Understanding the message logging paradigm for masking process crashes." Ph.D. Thesis, Department of Computer Science, Cornell University, Jan. 1996 (also available as Technical Report TR-96-1577).

[2] L. Alvisi, E.N. Elnozahy, S. Rao, S. A. Husain and A. Del Mel. "An analysis of communication-induced checkpointing." In *Proceedings of the Twenty Ninth International Symposium on Fault-Tolerant Computing*, Jun. 1999.

[3] A. W. Appel. "A runtime system." Technical Report CS-TR220-89, Department of Computer Science, Princeton University, 1989.

[4] O. Babaoglu and W. Joy. "Converting a swap-based system to do paging in an architecture lacking page-reference bits." In *Proceedings of the Symposium on Operating Systems Principles*, pp. 78—86, 1981.

[5] R. Baldoni, F. Quaglia, and B. Ciciani. "A *VP*-accordant checkpointing protocol preventing useless checkpoints." In *Proceedings of the IEEE Symposium on Reliable Distributed Systems,* pp. 61—67, Oct. 1998.

[6] J. P. Banâtre, M. Banâtre and G. Muller. "Ensuring data security and integrity with a fast stable storage." In *Proceedings of the fourth Conference on Data Engineering*, pp. 285—293, Feb. 1988.

[7] J.F. Bartlett. "A Non Stop Kernel." In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 22—29, 1981.

[8] A. Beguelin, E. Seligman and P. Stephan. "Application level fault tolerance in heterogeneous networks of workstations." In *Journal Parallel & Distributed Computing*, 43(2):147—155, Jun. 1997.

[9] B. Bhargava and S. R. Lian. "Independent checkpointing and concurrent rollback for recovery - An optimistic approach." In *Proceedings of the Symposium on Reliable Distributed Systems*, pp. 3—12, 1988.

[10] B. Bhargava, S.R. Lian and P.J. Leu. "Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms." In *Proceedings of the International Conference on Data Engineering*, pp. 182—189, Mar. 1990.

[11] A. Borg, W. Blau, W. Graetsch, F. Hermann, and W. Oberle. "Fault tolerance under UNIX." *ACM Transactions on Computing Systems*, 7(1):1—24, Feb 1989.

[12] N. S. Bowen and D. K. Pradhan. "Survey of checkpoint and rollback recovery techniques." Technical Report TR-91-CSE-17, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Jul. 1991.

[13] T.C. Bressoud and F.B. Schneider. "Hypervisor-based fault tolerance." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 1—11, Dec. 1995.

[14] D. Briatico A. Ciuffoletti, and L. Simoncini. "A distributed domino-effect free recovery algorithm." In *Proceedings of the IEEE International Symposium on Reliability, Distributed Software, and Databases,* pp. 207—215, Dec. 1984.

[15] M. Chandy and C.V. Ramamoorthy. "Rollback and recovery strategies for computer programs." In *IEEE Transactions on Computers*, vol. 21, pp. 546—556, Jun. 1972.

[16] M. Chandy and L. Lamport. "Distributed snapshots: Determining global states of distributed systems." In *ACM Transactions on Computing Systems*, 3(1):63—75, Aug. 1985.

[17] G. Deconinck, J. Vounckx, R. Lauwereins and J. A. Peperstraete. "Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback." In *Proceedings of the IASTED International Conference on Modeling and Simulation*, pp. 262—265, May 1993.

[18] E.N. Elnozahy. "Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication." Ph.D. Thesis, Rice University, Oct. 1993. Also available as Technical Report 93-212, Department of Computer Science, Rice University.

[19] E.N. Elnozahy. "How safe is probabilistic checkpointing?." In *Proceedings of the Twenty Eighth International Symposium on Fault-Tolerant Computing* (*FTCS-28*), Jun. 1998.

[20] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. "The performance of consistent checkpointing." In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, pp. 39—47, Oct. 1992.

[21] E.N. Elnozahy and W. Zwaenepoel. "On the use and implementation of message logging." In *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing* (*FTCS-24*), pp. 298—307, Jun. 1994.

[22] S.I. Feldman and C.B. Brown. "Igor: A system for program debugging via reversible execution." In *ACM SIPLAN Notices*, *Workshop on Parallel and Distributed Debugging*, 24(1):112—123, Jan. 1989.

[23] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. Bacon. "Transparent recovery of Mach applications." In *Proceedings of the Usenix Mach Workshop*, pp. 169—184, Oct. 1990.

[24] J. M. Hélary, A. Mostefaoui, R.H. Netzer, and M. Raynal. "Preventing useless checkpoints in distributed computations." In *Proceedings of the IEEE Symposium on Reliable Distributed Systems,* pp. 183—190, Oct. 1997.

[25] J. M. Hélary, A. Mostefaoui, and M. Raynal. "Virtual precedence in asynchronous systems: concepts and applications." In *Proceedings of the 11th workshop on distributed algorithms, WDAG'*97, LNCS press, 1997.

[26] Y. Huang and C. Kintala. "Software implemented fault tolerance: Technologies and experience." In *Proceedings of the Twenty Third International Symposium on Fault-Tolerant Computing* (*FTCS-23*), pp. 2—9, Jun. 1993.

[27] Y. Huang and Y-M. Wang. "Why optimistic message logging has not been used in telecommunication systems." In *Proceedings of the Twenty Fifth International Symposium on Fault-Tolerant Computing* (*FTCS-25*), pp. 459—463, Jun. 1995.

[28] D.B. Johnson. "Distributed system fault tolerance using message logging and checkpointing." Ph.D. Thesis, Rice University, Dec. 1989.

[29] D.B. Johnson and W. Zwaenepoel. "Sender-based message logging." In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing* (*FTCS-17*), pp. 14—19, Jun. 1987.

[30] D.B. Johnson and W. Zwaenepoel. "Recovery in distributed systems using optimistic message logging and checkpointing." *Journal of Algorithms*, 11(3):462—491, Sep. 1990.

[31] T. T-Y. Juang and S. Venkatesan. "Crash recovery with little overhead." In *Proceedings of the International Conference on Distributed Computing Systems*, pp. 454—461, May 1991.

[32] R. Koo and S. Toueg. "Checkpointing and rollback-recovery for distributed systems." In *IEEE Transactions on Software Engineering*, SE-13(1):23—31, Jan. 1987.

[33] T.H. Lai and T.H. Yang. "On distributed snapshots." In *Information Processing Letters*, vol. 25, pp. 153—158, 1987.

[34] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, 21(7):588—565, Jul. 1978.

[35] B.W. Lampson and H.E. Sturgis. "Crash recovery in a distributed data storage system." Technical Report, Xerox Palo Alto Research Center, April 1979.

[36] C.C. Li and W.K. Fuchs. "CATCH: Compiler-assisted techniques for checkpointing." In *Proceedings of the Twentieth International Symposium on Fault-Tolerant Computing* (*FTCS-20*), pp. 74—81, Jun. 1990.

[37] J. Mellor-Crummey and T. LeBlanc. "A software instruction counter." In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 78—86, Apr. 1989.

[38] C. Morin and I. Puaut. "A survey of recoverable distributed shared virtual memory systems." In *IEEE Transactions on Parallel and Distributed Systems*, 8(9):959—969, Sep. 1997.

[39] G. Muller, M. Hue and N. Peyrouz. "Performance of consistent checkpointing in a modular operating system: Results of the FTM Experiment." In *Lecture Notes in Computer Science: Dependable Computing, EDCC-1*, pp. 491—508, Oct. 1994.

[40] H-C Nam, J. Kim, SJ. Hong and S. Lee. "Probabilistic checkpointing." In *Proceedings of the Twenty Seventh International Symposium on Fault-Tolerant Computing* (*FTCS*-27), pp.48—57, Jun. 1997.

[41] R.B. Netzer and J. Xu. "Necessary and sufficient conditions for consistent global snapshots." In *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165—169, Feb. 1995.

[42] R. Pausch. "Adding input and output to the transactional model." Ph.D. Thesis, Carnegie Mellon University, August 1988.

[43] J.S. Plank. "Efficient checkpointing on MIMD architectures." Ph.D. Thesis, Princeton University, 1993.

[44] J.S. Plank, M. Beck, G. Kingsley and K. Li. "Lipckpt: Transparent checkpointing under UNIX." *In Proceedings of the USENIX Winter 1995 Technical Conference*, pp. 213—223, Jan. 1995.

[45] J.S. Plank and K. Li. "Faster checkpointing with $N + 1$ parity." In *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing (FTCS-24),* pp. 288—297, Jun. 1994.

[46] J. S. Plank, Jian Xu, R.B. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing." Technical Report CS-95-302, University of Tennessee at Knoxville, Aug. 1995.

[47] B. Randell. "System structure for software fault-tolerance." *IEEE Transactions on Software Engineering*, SE-1(2):220—232, Jun. 1975.

[48] S. Rao, L. Alvisi, and H.M. Vin. "The cost of recovery in message logging protocols." In *Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems* (*SRDS*), pp. 10—18, 1998.

[49] M. Ruffin. "KITLOG: A generic logging service." In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, pp. 139—148, Oct. 1992.

[50] D. L. Russell. "State restoration in systems of communicating processes." In *IEEE Transactions on Software Engineering*, SE-6(2):183—194, Mar. 1980.

[51] R.D. Schlichting and F.B. Schneider. "Fail-Stop processors: An approach to designing fault-tolerant computing systems." *ACM Transactions on Computer Systems*, vol. 1(3): 222—238, Aug. 1983.

[52] L.M. Silva. "Checkpointing mechanisms for scientific parallel applications." Ph.D. thesis, University of Coimbra, Portugal, Mar. 1997.

[53] A. Sistla and J. Welch. "Efficient distributed recovery using message logging." In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (*PODC*), pp. 223—238, Aug. 1989.

[54] J.H. Slye and E.N. Elnozahy. "Support for software interrupts in log-based rollback-recovery." In *IEEE Transactions on Computers,* 47(10), pp. 1113—1123, Oct. 1998.

[55] S.W. Smith and D.B. Johnson. "Minimizing timestamp size for completely asynchronous optimistic recovery with minimal rollback." In *Proceedings of the 15th Symposium on Reliable Distributed Systems* (*SRDS-15*), pp. 66—75, Oct. 1996.

[56] R. Strom and S. Yemini. "Optimistic recovery in distributed systems." *ACM Transactions on Computer Systems,* 3(3): 204—226, Aug. 1985.

[57] Y. Tamir and C.H. Séquin. "Error recovery in multicomputers using global checkpoints." In *Proceedings of the International Conference on Parallel Processing*, pp. 32—41, Aug. 1984.

[58] Z. Tong, R. Y. Kain and W. T. Tsai. "Rollback-recovery in distributed systems using loosely synchronized clocks." In *IEEE Transactions on Parallel and Distributed Systems*, 3(2):246—251, Mar. 1992.

[59] Y. M. Wang. "Space reclamation for uncoordinated checkpointing in message-passing systems." Ph.D. Thesis, University of Illinois Urbana-Champaign, Aug. 1993.

[60] Y. M. Wang. "Consistent global checkpoints that contain a set of local checkpoints." *IEEE Transactions on Computers*, 46(4):456—468, Apr. 1997.

[61] Y.M Wang and K. Fuchs. "Optimistic message logging for independent checkpointing in message passing systems." In *Proceedings of the IEEE Symposium on Reliable Distributed Systems,* pp. 147—154. Oct. 1992.

[62] Y. M. Wang, P. Y. Chung, I. J. Lin and W. K. Fuchs. "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems." In *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546—554, May 1995.

[63] Y. M. Wang, P. Y. Chung and W. K. Fuchs. "Tight upper bound on useful distributed system checkpoints." Technical ReportCRHC-95-16, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1995.

Bibliography

[1] A. Acharya and B. R. Badrinath. "Recording distributed snapshots based on causal order of message delivery." In *Information Processing Letters*, vol. 44, no. 6, Dec. 1992.

[2] A. Acharya and B. R. Badrinath. "Checkpointing distributed applications on mobile computers." In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Sep. 1994.

[3] M. Ahamad and L. Lin. "Using checkpoints to localize the effects of faults in distributed systems." In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pp. 2—11, 1989.

[4] M. Ahuja. "Repeated global snapshots in asynchronous distributed systems." Technical Report OSU-CISRC-8/89 TR40, The Ohio State University, Aug. 1989.

[5] M.S. Algudady and C.R. Das. "A cache-based checkpointing scheme for MIN-based multiprocessors." In *Proceedings of the International Conference on Parallel Processing*, pp. 497—500, 1991.

[6] L. Alvisi, B. Hoppe, and K. Marzullo. "Nonblocking and orphan-free message logging protocols." In *Proceedings of the Twenty Third International Symposium on Fault-Tolerant Computing* (*FTCS-23*), pp. 145—154, Jun. 1993.

[7] L. Alvisi and K. Marzullo. "Message logging: Pessimistic, optimistic and causal." In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, May 1995.

[8] L. Alvisi and K. Marzullo. "Deriving optimal checkpointing protocols for distributed shared memory architectures." In *Proceedings of the 1995 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing Systems* (*PODC*), pp. 263. Aug. 1995.

[9] L. Alvisi and K. Marzullo. "Trade-offs in implementing causal message logging protocols." In *Proceedings of the 1996 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing Systems* (*PODC*), pp. 58—67, 1996.

[10] L. Alvisi and K. Marzullo. "Message logging: Pessimistic, optimistic, causal and optimal." In *IEEE Transactions on Software Engineering*, 24(2):149—159, Feb. 1998.

[11] L. Alvisi, S. Rao, and H.M. Vin. "Low-overhead protocols for fault-tolerant file sharing." In *Proceedings of the IEEE 18th International Conference on Distributed Computing Systems*, pp. 452—461, May 1998.

[12] J.A. Anyanwu. "A reliable stable storage system for UNIX." In *Software—Practice and Experience*, 15(10):973—900, Oct. 1985.

[13] Y. Artsy and R. Finkel. "Designing a process migration facility: The Charlotte experience." *IEEE Computer*, pp. 47—56, Sep. 1989.

[14] N. Attig and V. Sander. "Automatic checkpointing of NQS batch jobs on CRAY UNICOS systems." In *Proceedings of the Cray User Group Meeting*, 1993.

[15] O. Babaoglu. "Fault-tolerant computing based on Mach." In *Proceedings of the Usenix Mach Workshop*, pp. 186—199, Oct. 1990.

[16] O. Babaoglu and K. Marzullo. "Consistent global states of distributed systems: Fundamental concepts and mechanisms." *Distributed Systems*, *Ed. S. Mullender*, *Addison-Wesley*, pp. 55—96, 1993.

[17] D.F. Bacon. "Transparent recovery in distributed systems." In *Operating Systems Review*, pp. 91—94, Apr. 1991.

[18] D.F. Bacon. "File system measurements and their application to the design of efficient operation logging algorithms." In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pp. 21—30, Oct. 1991.

[19] R. Baldoni, J-M. Hélary, A. Mostefaoui and M. Raynal. "A communication-induced checkpointing protocol that ensures rollback-dependency trackability." In *Proceedings of the International Symposium on Fault-Tolerant-Computing Systems*, pp. 68—77, Jun. 1997.

[20] R. Baldoni, J-M. Hélary, A. Mostefaoui and M. Raynal. "Adaptive checkpointing in message passing distributed systems." In *International Journal of Systems Science*, 28(11):1145—1161, vol. 28, no. 11, Nov. 1997.

[21] R. Baldoni, F. Quaglia, and P. Fornara. "An index-based checkpointing algorithm for autonomous distributed systems." *In Proceedings of the Sixteenth Symposium on Reliable Distributed Systems*, pp. 27—34, Oct. 1997.

[22] J. P. Banâtre, M. Banâtre and G. Muller. "Architecture of fault-tolerant multiprocessor workstations." In *Proceedings of the Workshop on Workstation Operating Systems*, pp. 20—24, 1989.

[23] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee and C. Morin. "An architecture for tolerating processor failures in shared-memory multiprocessors." Technical report No. 707-93, IRISA, Rennes, Mar. 1993.

[24] M. Banâtre, P. Heng, G. Muller and B. Rochard. "How to design reliable servers using fault-tolerant micro-kernel mechanisms." In *Proceedings of the USENIX Mach Symposium*, pp. 223—231, Nov. 1991.

[25] G. Barigazzi and L. Strigini. “Application-transparent setting of recovery points.” In *Proceedings of the Thirteenth International Symposium on Fault-Tolerant Computing Systems*, *FTCS*-13, pp. 48—55, 1983.

[26] M. Beck, J. S. Plank and G. Kingsley. “Compiler-assisted checkpointing.” Technical Report CS-94-269, Department of Computer Science, University of Tennessee at Knoxville, Dec. 1994.

[27] G. Beedubail, A. Karmarkar, A. Gurijala, W. Marti, and U. Pooch. “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*), pp. 142—148, 1995.

[28] K. Bhatia, K. Marzullo and L. Alvisi. “The relative overhead of piggybacking in causal message logging protocols.” In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp. 348—353, 1998.

[29] B. Bieker, G. Deoninck, E. Maehle and J. Vounckx. “Reconfiguration and checkpointing in massively parallel systems.” In *Proceedings of the 1st European Dependant Computing Conference*, EDCC-1, pp. 353—370, Oct. 1994.

[30] A. Borg, J. Baumbach, and S. Glazer. “A message system supporting fault tolerance.” In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pp. 90—99, Oct. 1983.

[31] N. S. Bowen and D. K. Pradhan. “Virtual checkpoints: Architecture and performance.” In *IEEE Transactions on Computers*, 41(5):516—525, May 1992.

[32] N. S. Bowen and D. K. Pradhan. “Processor- and memory-based checkpoint and rollback recovery.” In *IEEE Computer*, 26(2):22—32, Feb. 1993.

[33] T. Bressourd and F. Schneider. “Hypervisor-based fault tolerance.” In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[34] G. Cabillic, G. Muller and I. Puaut. “The performance of consistent checkpointing in distributed shared memory systems.” In *Proceedings of the 14th Symposium on reliable distributed systems*, SRDS-14, Sep. 1995.

[35] A.E. Campos and M.A. Castillo. “Checkpointing through garbage collection.” Technical report. Departamento de Ciencia de la Computación, Escuela de Ingeniería Pontificia Universidad Católica de Chile, Sep. 1996.

[36] J. Cao. “On correctness of distributed rollback recovery.” In *Proceedings of the 14th Australia Computer Science Conference*, pp. 39.1—39.10, Feb. 1991.

[37] J. Cao. “Efficient synchronous checkpointing in distributed systems.” In *Proceedings of the 15th Australia Computer Science Conference*, pp 165—179, Jan. 1992.

[38] G. Cao and M. Singhal. “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*, pp. 37—44, Aug. 1998.

[39] G. Cao and M. Singhal. “Low-cost checkpointing with mutable checkpoints in mobile computing systems.” In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pp. 464—471, May 1998.

[40] J. Cao and K. C. Wang. “Efficient synchronous checkpointing in distributed systems.” Technical Report 91/6, Department of Computer Science, James Cook University of North Queensland, Australia, Dec. 1991.

[41] J. Cao and K. C. Wang. “An abstract model of rollback recovery control in distributed systems.” *Operating Systems Review*, pp. 62—76, Oct. 1992.

[42] T. Cargill and B. Locanthi. “Cheap hardware support for software debugging and profiling.” In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 82—83, Oct. 1987.

[43] J.B. Carter, A. Cox, S. Dwarkadas, E.N. Elnozahy, D.B. Johnson, P. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel. “Network multicomputing using recoverable distributed shared memory.” In *Proceedings of COMPCON’93,* Nov. 1993.

[44] J. Casas, D. Clark, P. Galbiati and R. Konuru. “MIST: PVM with transparent migration and checkpointing.” Technical Report, Department of Computer Science, Oregon Graduate Institute of Science and Technology, May 1995.

[45] R. Chen and T.P. Ng. “Microkernel support for checkpointing.” *Open Forum*, Nov. 1992.

[46] R. Chen and T.P. Ng. “Building a fault-tolerant system based on Mach.” In *Proceedings of the USENIX Mach Workshop*, pp. 157—168, 1990.

[47] G-M. Chiu and C-R. Young. “Efficient rollback-recovery technique in distributed computing systems.” In *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 6, Jun. 1996.

[48] T. Chiueh. “Polar: A storage architecture for fast checkpointing.” In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, pp. 251—258, 1992.

[49] T.-C. Chiueh and P. Deng. “Evaluation of checkpoint mechanisms for massively parallel machines.” In *Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pp.370—379, Jun. 1996.

[50] M. Choy, H. Leong, and M.H. Wong. "On distributed object checkpointing and recovery." In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Aug. 1995.

[51] K-S. Chung, K-B. Kim, C-S. Hwang, J.G. Shon and H-C. Yu. "Hybrid checkpointing protocol based on selective sender-based message logging." In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pp. 788-793, Dec. 1997.

[52] A. Clematis, G. Dodero and V. Gianuzzi. "Process checkpointing primitives for fault tolerance: Definitions and examples." In *Microprocessors and Microsystems*, 16(1):15—23, 1992.

[53] A. Clematis. "Fault-tolerant programming for network based parallel computing." In *Microprocessing and Microprogramming*, vol. 40, pp. 765—768, 1994.

[54] F. Cristian and F. Jahanian. "A timestamp-based checkpointing protocol for long-lived distributed computations." In *Proceedings of The 10th Symposium on Reliable Distributed Systems, SRDS*, pp. 12—20, 1991.

[55] D. Cummings and L. Alkalaj. "Checkpoint/rollback in a distributed system using coarse-grained dataflow." *In Proceedings of the Twenty Fourth Annual International Symposium on Fault-Tolerant Computing, FTCS-24*, pp. 424—433, Jun. 1994.

[56] O.P. Damani and V. K. Garg. "How to recover efficiently and asynchronously when optimism fails." In *Proceedings of the 16th International Conference on Distributed Computing*, pp. 108—115, May 1996.

[57] G. Deconinck and R. Lauwereins. "User-triggered checkpointing: system-independent and scalable application recovery." In *Proceedings Second IEEE Symposium on Computer and Communications*, pp. 418—423, Jul. 1997.

[58] G. Deconinck, J. Vounckx, R. Lauwereins, and J. Peperstraete. "Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback." *International Journal of Modeling and Simulation*, 18(1):66—71, 1998.

[59] Z. Di. "Eliminating domino effect in backward error recovery in distributed systems." In *Proceedings of the 2nd International Conference on Computers and Applications*, pp. 243—248, 1987.

[60] A. Duda. "The effects of checkpointing on program execution time." In *Information Processing Letters*, no. 16, pp. 221—229, 1983.

[61] P.L. Ecuyer and J. Malefant. "Computing optimal checkpointing strategies for rollback and recovery systems." In *IEEE Transactions on Computers*, vol. 37, pp. 491—496, Apr. 1988.

[62] E.L. Ellenberger. "Transparent process rollback recovery: Some new techniques and a portable implementation." Masters Thesis, Department of Computer Science, Texas A&M University, Aug. 1995.

[63] B. Ellis. "A stable storage package." In *Proceedings of the USENIX Summer Conference*, pp. 209—212, 1985.

[64] E.N. Elnozahy. "Efficient fault-tolerance support for interactive distributed applications. Technical Report TR90-120, Department of Computer Science, Rice University, May 1990.

[65] E.N. Elnozahy. "Fault tolerance for clusters of workstations." In *Hardware and Software Architectures for Fault Tolerance.* M. Banâtre and P. Lee, Editors. Springer Verlag, Feb. 1994.

[66] E.N. Elnozahy. "On the relevance of communication costs of rollback-recovery protocols." In *Proceedings of the 1995 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing Systems* (*PODC*), Aug. 1995.

[67] E.N. Elnozahy and W. Zwaenepoel. "Replicated distributed processes in Manetho." In *Proceedings of the Twenty Second Annual International Symposium on Fault-Tolerant Computing*, *FTCS-22*, pp. 18—27, Jul. 1992.

[68] E.N. Elnozahy and W. Zwaenepoel. "Manetho, transparent rollback-recovery with low overhead, limited rollback and fast output commit." In *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 41(5):526—531, May 1992.

[69] E.N. Elnozahy and W. Zwaenepoel. "An integrated approach to fault tolerance." In *Proceedings of the Second Workshop on Management of Replicated Data,* pp. 82—85, Nov. 1992.

[70] C. J. Fidge. "Timestamps in message-passing systems that preserve the partial ordering." In *Proceedings of the 11th Australian Computer Science Conference*, pp. 55—66, Feb. 1988.

[71] M. J. Fischer, N.D. Griffeth and N. A. Lynch. "Global states of a distributed system." In *IEEE Transactions on Software Engineering*, SE-8(3):198—202, May 1982.

[72] T. M. Frazier and Y. Tamir. "Application-transparent error-recovery techniques for multicomputers." In *Proceedings of The Fourth Conferences on Hypercubes, Concurrent Computers, and Applications*, pp. 103—108, Mar. 1989.

[73] J. Gait. "A checkpointing page store for write-once optical disk." In *IEEE Transactions on Computers*, 39(1):2—9, Jan. 1990.

[74] S. Garg and K.F. Wong. "Improving the speed of a distributed checkpointing algorithm." In *Proceedings of the 6th International Conference on Parallel and Distributed Computing Systems*, Oct. 1993.

[75] E. Gelenbe. "On the optimum checkpointing interval." In *Journal of the ACM*, vol. 2, pp. 259—270, Apr. 1979.

[76] E. Gelenbe and D. Derochette. "Performance of rollback-recovery systems under intermittent failures." In *Communications of the ACM*, 21(6):493—499, Jun. 1978.

[77] S.T. Gregory and J.C. Knight. "On the provision of backward error recovery in production programming languages." In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing Systems*, *FTCS*-89, pp. 506—511, Jun. 1989.

[78] B. Groselj. "Bounded and minimum global snapshots." In *IEEE Parallel and Distributed Technology*, vol. 1, no.4, Nov. 1993.

[79] V. Hadzilacos. "An algorithm for minimizing rollback cost." In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 93—97, 1982.

[80] J. M. Hélary. "Observing global states of asynchronous distributed applications." *Lecture Notes in Computer Science*, Springer-Verlag, vol. 392, pp. 124—135, 1989.

[81] J. M. Hélary, A. Mostefaoui, R.H. Netzer, and M. Raynal. "Preventing useless checkpoints in distributed computations." In *Proceedings of the IEEE Symposium on Reliable Distributed Systems,* pp. 183—190, Oct. 1997.

[82] J. M. Hélary, A. Mostefaoui, and M. Raynal. "Virtual precedence in asynchronous systems: concepts and applications." In *Proceedings of the 11th workshop on distributed algorithms, WDAG'*97, LNCS press, 1997.

[83] J. M. Hélary, A. Mostefaoui, and M. Raynal. "Communication-induced determination of consistent snapshots." In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing,* pp. 208—217, Jun. 1998.

[84] C. E. Hewitt. "Checkpoint and recovery in ACTOR systems." Technical Report, MIT Artificial Intelligence Laboratory, 1980.

[85] H. Higaki, K. Shima, T. Tachikawa, and M. Takizawa. "Checkpoint and rollback in asynchronous distributed systems." *In Proceedings IEEE INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 998—1005, Apr. 1997.

[86] H. Higaki and M. Takizawa. "Checkpoint-recovery protocol for reliable mobile systems." In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp. 93—99, Oct. 1998.

[87] S. Israel and D. Morris. "A non-intrusive checkpointing protocol." In *Proceedings of the Phoenix Conference on Communications and Computers*, pp. 413—421, Mar. 1989.

[88] P. Jalote. "Fault-tolerant processes." In *Distributed Computing*, vol. 3, pp. 187—195, 1989.

[89] G. Janakiraman and Y. Tamir. "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers." In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 42—51, Oct. 1994.

[90] B. Janssens and W.K. Fuchs. "Relaxing consistency in recoverable distributed shared memory." In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pp. 155—163, Jun. 1993.

[91] B. Janssens and W.K. Fuchs. "Reducing interprocessor dependence in recoverable distributed shared memory." In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 34—41, Oct. 1994.

[92] D. P. Jasper. "A discussion of checkpoint restart." In *Software Age*, Oct. 1969.

[93] K. Jeong and D. Shahsa. "Plinda 2.0: A transactional/checkpoint approach to fault-tolerant Linda." In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 96—105, Oct. 1994.

[94] D. B. Johnson. "Efficient transparent optimistic rollback recovery for distributed application programs." In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pp. 86—95, Oct. 1993.

[95] D. B. Johnson and W. Zwaenepoel. "Output-driven distributed optimistic message logging and checkpointing." Technical Report TR90-118, Department of Computer Science, Rice University, May 1990.

[96] D.B. Johnson and W. Zwaenepoel. "Recovery in distributed systems using optimistic message logging and checkpointing." In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing Systems*, *PODC*-88, pp. 171—181, Aug. 1988.

[97] D. B. Johnson and W. Zwaenepoel. "Transparent optimistic rollback recovery." In *Operating Systems Review*, pp. 99—102, Apr. 1991.

[98] T. Juang and S. Venkatesan. "Crash recovery with little overhead." In *Proceedings of the 11th International Conference on Distributed Computing Systems* (*ICDCS-11*), pp. 454—461, May 1991.

[99] M. F. Kaashoek, R. Michiels, H. E. Bal and A. S. Tanenbaum. "Transparent fault-tolerance in parallel Orca programs." In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, pp. 297—312, Mar. 1992.

[100] S. Kambhalta and J. Walpole. “Recovery with limited replay: Fault-tolerant processes in Linda.” In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 715—718, 1990.

[101] K. Kant. “A model for error recovery with global checkpointing.” *Information Sciences*, no. 30, pp. 58—68, 1978.

[102] S. Kanthadai and J. L. Welch. “Implementation of recoverable distributed shared memory by logging writes.” In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pp. 27—30, May 1996.

[103] A.M. Kermarrek, G. Cabillic, A. Gefflaut, C. Morin and I. Puaut. “A recoverable distributed shared memory integrating coherence and recoverability.” In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pp. 289—298, Jul. 1995.

[104] J. L. Kim and T. Park. “An efficient protocol for checkpointing recovery in distributed systems.” In *IEEE Transactions on Parallel and Distributed Systems*, 4(8):955—960, Aug. 1993.

[105] K. H. Kim. “Approaches to mechanization of the conversation scheme based on monitors.” In *IEEE Transactions on Software Engineering*, SE-8(3):189—197, May 1982.

[106] K. H. Kim. “Programmer-transparent coordination of recovering concurrent processes: Philosophy and rules for efficient implementation.” In *IEEE Transactions on Software Engineering*, SE-14(6):189—197, Jun. 1988.

[107] K. H. Kim and J. H. You. “A highly decentralized implementation model for the Programmer-Transparent Coordination (PTC) scheme for cooperative recovery.” In *Proceedings of the Twentieth International Symposium on Fault-Tolerant Computing System*, pp. 282—289, 1990.

[108] K. H. Kim, J. H. You and A. Abouelnaga. “A scheme for coordinated execution of independently designed recoverable distributed processes.” In *Proceedings of the Sixteenth International Symposium on Fault-Tolerant Computing System*, pp. 130—135, 1986.

[109] Y. Kim, J.S. Plank and J.J. Dongarra. “Fault-tolerant matrix operations using checksum and reverse computation.” In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation,* Oct. 1996.

[110] Y. Kim, J.S. Plank and J.J. Dongarra. “Fault-tolerant matrix operations for network of workstations using multiple checkpointing.” In *Proceedings of HPC Asia’97*, *High Performance Computing in the Information Superhighway*, pp. 460—465, Apr. 1997.

[111] B. A. Kingsbury and J. T. Kline. “Job and process recovery in a UNIX-based operating system.” In *Usenix Association, Winter Conference Proceedings*, pp.355—364, Jan. 1989.

[112] A.C. Klaiber and H.M. Levy. “Crash recovery for scientific applications.” In *Proceedings of the International Conference on Parallel and Distributed Systems*, 1993.

[113] C.M. Krishna, G. Kang and Y. Lee. “Optimization criteria for checkpoint placement.” In *Communications of the ACM,* 27(10):1008—1012, Oct. 1984.

[114] P. Krishna, N.T. Vaidya and D.K. Pradhan. “Recovery in multicomputers with finite error detection latency.” In *Proceedings of the 23rd International Conference on Parallel Processing*, Aug. 1994.

[115] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems.” In *ACM Transactions on Programming Languages and Systems*, 6(2):254—280, Apr. 1984.

[116] C.R. Landau. “The checkpoint mechanism in KeyKOS.” In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, Sep. 1992.

[117] B. Lee, T. Park, H. Yeom, and Y. Cho. “An efficient algorithm for causal message logging.” In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp. 19—25, 1998.

[118] J. Leon, A.L. Ficher, and P. Steenkiste. “Fail-safe PVM: A portable package for distributed programming with transparent recovery.” Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[119] H.V. Leong and D. Agrawal. “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*), pp. 227—234, 1994.

[120] P.-J Leu and B. Bhargava. “Concurrent robust checkpointing and recovery in distributed systems.” In *Proceedings of the International Conference on Data Engineering*, pp. 154—163, Feb. 1988.

[121] K. Li, J.F. Naughton and J.S. Plank. “Real-time, concurrent checkpoint for parallel programs.” In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pp. 79—88, Mar. 1990.

[122] K. Li, J.F. Naughton and J.S. Plank. “Checkpointing multicomputer applications.” In *Proceedings of the 10th Symposium on Reliable Distributed Systems,* pp. 1—10, Oct. 1991.

[123] K. Li, J.F. Naughton and J.S. Plank. “An efficient checkpointing method for multicomputers with wormhole routing.” In *International Journal of Parallel Programming*, 20(3):159—180, Jun. 1992.

[124] W-J Li. And J-J Tsay. “Checkpointing message-passing interface (MPI) parallel programs.” In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pp. 147—152, 1997.

[125] L. Lin and M. Ahamad. “Checkpointing and rollback-recovery in distributed object based systems.” In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pp. 97—104, 1990.

[126] T-H. Lin and K.G. Shin. “Damage assessment for optimal rollback-recovery.” In *IEEE Transactions on Computers*, 47(5):603—613, May 1998.

[127] M. Litzkow and M. Solomon. “Supporting checkpointing and process migration outside the UNIX kernel.” In *Usenix Winter 1992 Technical Conference*, pp. 283—290, Jan. 1992.

[128] J. Long, W.K. Fuchs and J.A. Abraham. “Forward recovery using checkpointing in parallel systems.” In *Proceedings of the 19th International Conference on Parallel Processing*, pp. 272—275, Aug. 1990.

[129] J. Long, W.K. Fuchs and J.A. Abraham. “Implementing forward recovery using checkpointing in distributed systems.” In *Proceedings of the International Conference on Dependable Computing for Critical Applications*, DCCA, pp. 20—27, 1991.

[130] J. Long, W.K. Fuchs and J.A. Abraham. “Compiler-assisted static checkpoint insertion.” In *Proceedings of the Twenty Second Annual International Symposium on Fault-Tolerant Computing*, *FTCS-22*, pp. 58—65, Jul. 1992.

[131] A. Lowry, J.R. Russell and A.P. Goldberg. “Optimistic failure recovery for very large networks.” In *Proceedings of the Symposium on Reliable Distributed Systems*, pp. 66—75, 1991.

[132] K.I. Mandelberg and V.S. Sunderam. “Process migration in UNIX networks.” In *Proceedings of the Usenix Winter Technical Conference*, pp. 357—364, 1988.

[133] D. Manivannan and M. Singhal. “A low-overhead recovery technique using synchronous checkpointing.” In *Proceedings of International Conference on Distributed Computing Systems*, pp.100—107, May 1996.

[134] D. Manivannan, R. H. Netzer, and M. Singhal. “Finding consistent global checkpoints in a distributed computation.” In *IEEE Transactions on Parallel & Distributed Systems*, 8(6):623—627, Jun. 1997.

[135] F. Mattern. “Virtual time and global states of distributed systems.” In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pp. 215—226, Oct. 1988.

[136] J.A. McDermid. “Checkpointing and error recovery in distributed systems.” In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pp. 271—282, 1981.

[137] P. M. Merlin and B. Randell. “State restoration in distributed systems.” In *Proceedings of the Eighth International Symposium on Fault-Tolerant Computing Systems*, pp. 129—134, Jun. 1978.

[138] J.R. Mitchell and V.K. Garg. “A non-blocking recovery algorithm for causal message logging.” In *Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems*, pp. 3—9, 1998.

[139] A. Mostefaoui and M. Raynal. “Efficient message logging for uncoordinated checkpointing protocols.” In *Dependable Computing—EDCC-2*, *the Second European Dependable Computing Conference Proceedings*, *Springer-Verlag*, pp.353—364, Oct. 1996.

[140] G. Muller, M. Hue and N. Peyrouz. “Performance of consistent checkpointing in a modular operating system: results of the FTM experiment.” In *Proceedings of the First European Dependable Computing Conference* (*EDCC-1*) , pp. 491—508, Oct. 1994.

[141] G. Muller, M. Banâtre , N. Peyrouz and B. Rochat. “Lessons from FTM: an experiment in design and implementation of a low-cost fault-tolerant system.” In *IEEE Transactions on Reliability*, 45(2):332—340, Jun. 1996.

[142] E. Nett, R. Kroger and J. Kaiser. “Implementing a general error recovery mechanism in a distributed operating system.” In *Proceedings of the Sixteenth International Symposium on Fault-Tolerant Computing* (*FTCS-16*), pp.124—129, 1986.

[143] R.B. Netzer and B.P. Miller. “Optimal tracing and replay for debugging message-passing parallel programs.” In *Proceedings of Supercomputing*’92, pp. 502—511, Nov. 1992.

[144] R.B. Netzer and M.H. Weaver, “Optimal tracing and incremental reexecution for Debugging Long-Running Programs.” In *SIGPLAN ’94: Conference on Programming Language Design and Implementation* (*PLDI*), pp. 313—325, Jun. 1994.

[145] R.B. Netzer and J. Xu. “Adaptive message logging for incremental program replay.” In *IEEE Parallel and Distributed Technology*, 1(4):32—39, Nov. 1993.

[146] R.B. Netzer and J. Xu. “Replaying distributed programs without message logging.” In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing* (*HPDC*), pp. 137—147, Aug. 1997.

[147] N. Neves, M. Castro and P. Guedes. “A checkpoint protocol for an entry consistent shared memory system.” In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, Aug. 1994.

[148] N. Neves and W. K. Fuchs. “Using time to improve the performance of coordinated checkpointing.” In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, IPDS’96, pp. 282—291, Sep. 1996.

[149] N. Neves and W. K. Fuchs. "RENEW: A tool for fast and efficient implementation of checkpoint protocols." In *Proceedings of the Twenty Seventh IEEE Fault-Tolerant Computing Symposium,* Jun. 1998

[150] N. Neves and W. K. Fuchs. "Coordinated checkpointing without direct coordination." In *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (*IPDS'98*), pp. 23—31, Sep. 1998.

[151] V. Nicola. "Checkpointing and the modeling of program execution time." In *Software Fault Tolerance*, M. Lyu Ed., John Wiley and Sons, 1995.

[152] L. L. Peterson, N. C. Buchholz and R. D. Schlichting. "Preserving and using context information in interprocess communication." In *ACM Transaction on Computing Systems*, 7(3):217—246, Aug. 1989.

[153] S. L. Peterson and P. Kearns. "Rollback based on vector time." In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pp. 68—77, Oct. 1993.

[154] J. S. Plank. "Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques." In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, pp. 76—85, Oct. 1996.

[155] J. S. Plank, M. Beck and G. Kingsley. "Compiler-assisted memory exclusion for fast checkpointing." In *IEEE Technical Committee on Operating Systems Newsletter, Special Issue on Fault Tolerance*, pp. 62—67, Dec. 1995.

[156] J. S. Plank, Y. Chen, K. Li, M. Beck and G. Kingsley. "Memory exclusion: Optimizing the performance of checkpointing systems." Technical Report UT-CS-96-335, University of Tennessee, Aug. 1996.

[157] J. S. Plank and W.R. Elwasif. "Experimental assessment of workstation failures and their impact on checkpointing systems." In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing,* pp. 48—57, Jun. 1998.

[158] J. S. Plank, K. Li, and M.A. Puening. "Diskless checkpointing." *IEEE Transactions on Parallel & Distributed Systems*, 9(10):972—986, Oct. 1998.

[159] J. S. Plank, Y. Kim and J.J. Dongarra. "Algorithm-based diskless checkpointing for fault-tolerant matrix computations." In *Proceedings of the Twenty Fifth International Symposium on Fault-Tolerant Computing Systems*, pp. 351—360, Jun. 1995.

[160] J. S. Plank, K. Youngbae and J. J. Dongara. "Fault-tolerant matrix operations for networks of workstations using diskless checkpointing." In *Journal of Parallel & Distributed Computing*, 43(2):125—138, Jun. 1997.

[161] M. Powell and D. Presotto. "Publishing: A reliable broadcast communication mechanism." In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pp. 100—109, Oct. 1993.

[162] D.K. Pradhan and N. Vaidya. "Roll-forward checkpointing scheme: Concurrent retry with non-dedicated spares." In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 166—174, Jul. 1992.

[163] D.K. Pradhan and N. Vaidya. "Roll-forward and rollback-recovery: Performance-reliability trade-off." In *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing Systems*, pp. 186—195, Jun. 1994.

[164] B. Ramamurthy, S. Upadhyaya, and J. Bhargava. "Design and analysis of a hardware-assisted checkpointing and recovery scheme for distributed applications." In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp. 84—90, Oct. 1998.

[165] B. Ramamurthy, S. Upadhyaya and R.K. Iyer. "An object-oriented testbed for the evaluation of checkpointing and recovery systems." In *Proceedings of the Twenty Seventh International Symposium on Fault-Tolerant Computing* (*FTCS-27*), pp. 194—203, Jun. 1997.

[166] P. Ramanathan and K. G. Shin. "Checkpointing and rollback recovery in a distributed system using common time base." In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, SRDS-7, pp. 13—21, 1988.

[167] P. Ramanathan and K. G. Shin. "Use of common time base for checkpointing and rollback recovery in a distributed system." In *IEEE Transactions on Software Engineering*, SE-19(6):571—583, Jun. 1993.

[168] S. Rangarajan, S. Garg and Y. Huang. "Checkpoints-on-demand with active replication." In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp. 75—83, Oct. 1998.

[169] B. Ramkumar and V. Strumpen. "Portable checkpointing for heterogeneous architectures." In *Proceedings of the Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*, pp. 58—67, Jun. 1997.

[170] S. Rao, L. Alvisi and H. Vin. "Egida: An extensible toolkit for low-overhead fault tolerance." In *Proceedings of the Twenty Ninth International Symposium on Fault-Tolerant Computing*, Jun. 1999.

[171] G. G. Richard III and M. Singhal. "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory." In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pp. 58—67, Oct. 1993.

[172] M. Russinovich and B. Cogswell. "Replay for concurrent non-deterministic shared memory applications." In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 258—266, May 1996.

[173] M. Russinovich, Z. Segall, and D.P. Siewiorek. "Application transparent fault management in fault-tolerant Mach." In *Proceedings of the Twenty Third International Symposium on Fault-tolerant Computing* (*FTCS-23*), pp.10—19, Jun. 1993.

[174] R. Schwarz and F. Mattern. "Detecting causal relationships in distributed computations: in search of the Holy Grail." In *Distributed Computing*, vol. 7, pp. 149—174, 1994.

[175] E. Seligman and A. Beguelin. "High-level fault tolerance in distributed programs." Technical Report CMU-CS-94-223, Department of Computer Science, Carnegie Mellon University, Dec. 1994.

[176] D.D. Sharma and D. K. Pradhan. "An efficient coordinated checkpointing scheme for multicomputers." In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Jun. 1994.

[177] L.M. Silva and J.G. Silva. "Global checkpointing for distributed programs." In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pp. 155—162, Oct. 1992.

[178] L.M. Silva and J.G. Silva. "Integrating a checkpointing and rollback-recovery algorithm with a causal order protocol." In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pp. 523—540, May 1994.

[179] L.M. Silva and J.G. Silva. "Checkpointing pipeline applications." In *Proceedings of the 1994 World Transputer Congress*, pp. 497—512, Sep. 1994.

[180] L.M. Silva and J.G. Silva. "On the optimum recovery of distributed programs." In *Proceedings of the 20th EUROMICRO Conference*, pp. 704—711, Sep. 1994.

[181] L.M. Silva and J.G. Silva. "A checkpointing facility for a heterogeneous DSM system." In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, pp. 554—559, Sep. 1996.

[182] L.M. Silva and J.G. Silva. "Avoiding checkpoint contamination in parallel system." In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pp. 364—369, Jun. 1998.

[183] L.M. Silva and J.G. Silva. "An experimental study about diskless checkpointing." In *Proceedings of the 24th EUROMICRO Conference*, pp. 395—402, Aug. 1998

[184] L.M. Silva and J.G. Silva. "System-level versus user-defined checkpointing." In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp. 68—74, Oct. 1998.

[185] L.M. Silva, J.G. Silva and S. Chapple. "Portable transparent checkpointing for distributed shared memory." In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, HPDC-5, pp. 422-431, Aug. 1996.

[186] L.M. Silva, J.G. Silva, S. Chapple and L. Clarke. "Portable checkpointing and recovery." In *Proceedings of the 4th International Symposium on High-Performance Distributed Computing*, HPDC-4, pp. 188—195, Aug. 1995.

[187] L.M. Silva, V.N. Tavora and J.G. Silva. "Mechanisms of file-checkpointing for UNIX applications." In *Proceedings of the 14th IASTED Conference on Applied Informatics*, pp. 358—361, Feb. 1996.

[188] L.M. Silva, B. Veer and J.G. Silva. "Checkpointing SPMD applications on transputer networks." In *Proceedings of the Scalable High-Performance Computing Conference*, SHPCC94, pp. 694—701, May 1994.

[189] A. Sinha, P.K. Das and A. Chaudhuri. "Checkpointing and recovery in a pipeline of transputers." In *Proceedings of Euromicro'92*, pp. 141—148, 1992.

[190] J. H. Slye. "Adding support for software interrupts in log-based rollback-recovery protocols." Master's Thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, Dec. 1996.

[191] J. H. Slye and E.N. Elnozahy. "Supporting nondeterministic execution in fault-tolerant systems." In *Proceedings of the Twenty Sixth International Symposium on Fault-Tolerant Computing* (*FTCS-26*), Jun. 1996.

[192] J. M. Smith and J. Ioannidis. "Implementing remote fork() with checkpoint/restart." In *IEEE Technical Committee on Operating Systems Newsletter*, pp. 12—16, Feb. 1989.

[193] H. .M. Soliman and A.S. Elmaghraby. "An analytical model for hybrid checkpointing in time warp distributed simulation." *IEEE Transactions on Parallel & Distributed Systems*, 9(10):947—951, Oct. 1998.

[194] M. Spezialetti and P. Kearns. "Efficient distributed snapshots." In *Proceedings of the International Conference on Distributed Computing Systems*, pp. 382—388, 1986.

[195] K-F. Ssu, W.K. Fuchs. "PREACHES—portable recovery and checkpointing in heterogeneous systems." In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pp. 38—47, Jun. 1998.

[196] R. Stainov. "An asynchronous checkpointing service." In *Microprocessing and Microprogramming*, Vol. 31, pp. 117—120, Apr. 1991.

[197] M. Staknis. "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*, pp. 96—102, Apr. 1989.

[198] G. Stellner. "Consistent checkpoints of PVM applications. In *Proceedings of the First European PVM User Group Meeting*, 1994.

[199] G. Stellner. "CoCheck: Checkpointing and process migration for MPI." In *Proceedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[200] R. E. Strom, D. F. Bacon and S. A. Yemini. "Volatile logging in n-fault-tolerant distributed systems." In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing Systems*, pp. 44—49, 1988.

[201] R. E. Strom, S.A. Yemini and D. F. Bacon. "A recoverable object store." In *Proceedings of the Hawaii International Conference on System Sciences*, pp. II-215—II221, Jan. 1988.

[202] G. Suri, Y. Huang, Y. M. Wang, W. K. Fuchs and C. Kintala. "An implementation and performance measurement of the progressive retry technique." In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pp. 41—48, Apr. 1995.

[203] G. Suri, B. Janssens, and W.K. Fuchs. "Reduced overhead logging for rollback recovery in distributed shared memory." In *Proceedings of the Twenty Fifth International Symposium on Fault-Tolerant Computing* (*FTCS-25*), pp. 279—288, Jun. 1995.

[204] V.-O. Tam and M. Hsu. "Fast recovery in distributed shared virtual memory systems." In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 38—45, May 1990.

[205] Y. Tamir and T. M. Frazier. "Application-transparent process-level error recovery for multicomputers." In *Proceedings of the Hawaii International Conferences on System Sciences-22*, pp. 296—305, Jan. 1989.

[206] Y. Tamir and T. M. Frazier. "Error-recovery in multicomputers using asynchronous coordinated checkpointing." Technical Report CSD-010066, University of California, Los Angeles, Sep. 1991.

[207] Y. Tamir and E. Gafni. "A software-based hardware fault tolerance scheme for multicomputers." In *Proceedings of the International Conference on Parallel Processing*, pp. 117—120, Aug. 1987.

[208] K. Tanaka. H. Higaki and M. Takizawa. "Object-based checkpoints in distributed systems." In *Computer Systems Science & Engineering*, 13(3):179—185, May 1998.

[209] K. Tanaka K and M. Takizawa. "Distributed checkpointing based on influential messages." In *Proceedings of the 1996 International Conference on Parallel and Distributed Systems*, pp. 440—447, Jun. 1996.

[210] D. J. Taylor and M.L. Wright. "Backward error recovery in a UNIX environment." In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, pp. 118—123, Jun. 1986.

[211] S. Thanawastian, R. S. Pamula and Y. L. Varol. "Evaluation of global checkpoint rollback strategies for error recovery in concurrent processing systems." In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing Systems*, pp. 246—251, 1986.

[212] Z. Tong, R.Y. Kain and W.T. Tsai. "A lower overhead checkpointing and rollback recovery scheme for distributed systems." In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pp. 12—20, Oct. 1989.

[213] J. Tsai, S.Y. Kuo, and Y-M. Wang. "Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability." *IEEE Transactions on Parallel & Distributed Systems*, 9(10):963—971, Oct. 1998.

[214] K. Tsuruoka, A. Kaneko and Y. Nishihara. "Dynamic recovery schemes for distributed processes." In *Proceedings of the IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, pp. 124—130, 1981.

[215] P. Tullmann, J. Lepreau, B. Ford and M. Hibler. "User-level checkpointing through exportable kernel state." In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems*, pp. 85—88, Oct. 1996.

[216] N. H. Vaidya. "Dynamic cluster-based recovery: Pessimistic and optimistic schemes." Technical Report 93-027, Department of Computer Science, Texas A&M University, May 1993.

[217] N. H. Vaidya. "A case of two-level distributed recovery schemes." In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS'95*), pp. 64—73, May 1995.

[218] N. H. Vaidya. "On staggered checkpointing." In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pp.572—580, Oct. 1996.

[219] S. Venkatesan. "Message-optimal incremental snapshots." In *Proceedings of the International Conference on Distributed Computing Systems*, pp. 53—60, 1989.

[220] S. Venkatesan. "Optimistic crash recovery without changing application messages." In *IEEE Transactions on Parallel and Distributed Systems*, 8(3):263—271, Mar. 1997.

[221] K. Venkatesh, T. Radakrishnan, and H.L. Li. "Optimal checkpointing and local recording for domino-free rollback-recovery." *Information Processing Letters,* vol. 25, pp. 295—303, 1987.

[222] Y. M. Wang. "Reducing message logging overhead for log-based recovery." In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 1925—1928, May 1993.

[223] Y. M. Wang. "The maximum and minimum consistent global checkpoints and their applications." In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, Sep. 1995.

[224] Y. M. Wang, P. Y. Chung and W. K. Fuchs. "Tight upper bound on useful distributed system checkpoints." Technical ReportCRHC-95-16, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1995.

[225] Y. M. Wang, E. Chung, Y. Huang, and E.N. Elnozahy. "Integrating checkpointing with transaction processing." In *Proceedings of the Twenty Seventh International Symposium on Fault-Tolerant Computing* (*FTCS-27*), pp. 304—308, Jun. 1997.

[226] Y. M. Wang, O. P. Damani and V. K. Garg. "Distributed recovery with K-optimistic logging." In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pp. 60—67, May 1997.

[227] Y. M. Wang and K. Fuchs. "Scheduling message processing for reducing rollback propagation." In *Proceedings of the Twenty Second International Symposium on Fault-Tolerant Computing* (*FTCS-22*), pp. 204—211, Jul. 1992.

[228] Y. M. Wang and K. Fuchs. "Lazy checkpoint coordination for bounding rollback propagation." In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pp. 78—85, Oct. 1993.

[229] Y. M. Wang and W. K. Fuchs. "Optimal message log reclamation for uncoordinated checkpointing." In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994.

[230] Y. M. Wang, Y. Huang and W. K. Fuchs. "Progressive retry for software error recovery in distributed systems." In *Proceedings of the Twenty Third International Symposium on Fault-Tolerant Computing Systems*, *FTCS*-23, pp.138—144, Jun. 1993.

[231] Y. M. Wang, Y. Huang, and C. Kintala. "Progressive retry for software failure recovery in message passing applications." *IEEE Transactions on Computers*, vol. 46(10):1137—1141, Oct. 1997.

[232] Y. M. Wang, Y. Huang, K.P. Vo, P.Y. Chung, and C. Kintala. "Checkpointing and its applications." In *Proceedings of the Twenty Fifth International Symposium on Fault-Tolerant Computing* (*FTCS-25*), pp. 22-31, Jun. 1995.

[233] Y. M. Wang, A. Lowry and W. K. Fuchs. "Consistent global checkpoints based on direct dependency tracking." In *Information Processing Letters*, vol. 50, no. 4, pp. 223—230, May 1994.

[234] X. Wei and J. Ju. "A consistent checkpointing algorithm with shorter freezing time." *Operating Systems Reviews*, 32(4):pp. 70—76, Oct. 1998.

[235] Z. Wŏjcik and B. E. Wŏjcik. "Fault-tolerant distributed computing using atomic send receive checkpoints." In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 215—222, 1990.

[236] K.F. Wong and M. Franklin. "Checkpointing in distributed computing systems." In *Journal of Parallel & Distributed Computing*, pp. 67—75, May 1996.

[237] W. G. Wood. "A decentralized recovery control protocol." In *Proceedings of the Eleventh International Conference on Fault-Tolerant Computing Systems*, pp. 159—164, 1981.

[238] W. G. Wood. "Recovery control of communicating processes in a distributed system." In *Reliable Computer Systems, Ed. S. K. Shrivastava, Springer–Verlag*, pp. 448—473, 1985.

[239] K. L. Wu and W. K. Fuchs. "Recoverable distributed shared virtual memory." In *IEEE Transactions on Computers*, 39(4):460—469, Apr. 1990.

[240] D. S. Wyner. "A technique for optimizing the performance of a checkpoint restart system." In *Proceedings of the Canadian Computer Conference*, Montreal, pp. 201—212, Jun. 1972.

[241] J. Xu and R. H. B. Netzer. "Adaptive independent checkpointing for reducing rollback propagation." In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pp. 754—761, Dec. 1993.

[242] J. Xu, R.B. Netzer, and M. Mackey. "Sender-based message logging for reducing rollback propagation." In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pp. 602—609, 1995.

[243] J. W. Young. "A first order approximation to the optimum checkpoint interval." In *Communications of the ACM*, Vol. 17, No. 9, Sep. 1974.

[244] F. Zambonelli. "Distributed checkpoint algorithms to avoid rollback propagation." In *Proceedings of the 24th EUROMICRO Conference*, pp. 403—410, Aug. 1998.

[245] A. Ziv and J. Bruck. "Efficient checkpointing over local area networks." In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 30—35, Jun. 1994.

[246] A. Ziv and J. Bruck. "An on-line algorithm for checkpoint placement." In *IEEE Transactions on Computers*, 46(9):976—985, Sep. 1997.

[247] M. Zweiacker. "Fault-tolerant CORBA using checkpointing and recovery." In *Comtec*, 75(8):20—25, 1997.