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

E.N. Elnozahy D.B. Johnson Y.M. Wang

October 3, 1996

CMU-CS-96-181

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A version of this paper has been submitted for publication in ACM Computing Surveys.

E.N. Elnozahy was supported in part by the National Science Foundation under Research Initiation Award CCR-9410116 and CAREER Award CCR-9502933, and by the Defense Advanced Research Project Agency under contract DABT63-93-C0054.

D.B. Johnson was supported in part by the Defense Advanced Research Projects Agency under contract number DABT63-93-C-9954, and by the National Science Foundation under CAREER Award NCR-9502725.

Y.M. Wang is with AT&T Laboratories.

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

Abstract

The problem of rollback-recovery in message-passing systems has undergone extensive study. In this survey, we review rollback-recovery techniques that do not require special language constructs, and classify them into two primary categories. *Checkpoint-based rollback-recovery* relies solely on checkpointed states for system state restoration. Depending on when checkpoints are taken, existing approaches can be divided into uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing. *Log-based rollback-recovery* uses checkpointing and message logging. The logs enable the recovery protocol to reconstruct the states that are not checkpointed. There are three different log-based approaches, namely, pessimistic logging, optimistic logging and causal logging. We identify a set of desirable properties of rollback-recovery protocols, and compare different approaches with respect to these properties. Log-based rollback-recovery protocols generally rely on the assumption of piecewise determinism and pay additional overhead to allow faster output commits and more localized recovery. We present research issues under each approach, and review existing solutions to address them. We also present implementation issues of checkpointing and message logging.

1 Introduction

Rollback-recovery achieves fault tolerance by periodically saving the state of a process during failure-free execution, and restarting from a saved state upon a failure to reduce the amount of lost work. The saved process state is called a *checkpoint*, and the procedure of restarting from previously checkpointed state is called *rollback-recovery*. A checkpoint can be saved on either stable storage or the volatile storage of another process, depending on the failure scenarios to be tolerated. For long-running scientific applications, checkpointing and rollback-recovery can be used to minimize the total execution times in the presence of failures. For mission-critical service-providing applications, checkpointing and rollback-recovery can be used to improve service availability by providing faster recovery to reduce service down time.

Rollback-recovery in message-passing systems is complicated by the issue of *rollback propagation* due to interprocess communications. When the sender of a message m rolls back to a state before sending m, the receiver process must also roll back to a state before m's receipt; otherwise, the states of the two processes would be *inconsistent* because they would show that message m was not sent but has been received, which is impossible in any correct failure-free execution. Under some scenarios, cascading rollback propagation may force the system to restart from the initial state, losing all the work performed before a failure. This unbounded rollback is called the *domino effect* [144]. The possibility of the domino effect is highly undesirable because all checkpoints taken may turn out to be useless for protecting an application against losing all useful work upon a failure.

In a message-passing system, if each participating process takes its checkpoints independently then the system is susceptible to the domino effect. This approach is called *uncoordinated checkpointing* or *independent checkpointing*. One way to avoid the domino effect is to perform *coordinated checkpointing*: the processes in a system coordinate their checkpoints to form a system-wide consistent state. Such a consistent set of checkpoints can then be used to bound the rollback propagation. Alternatively, *communication-induced checkpointing* forces each process to take checkpoints based on some application messages it receives from other processes. This approach does not require system-wide coordination and therefore may scale better. The checkpoints are taken such that a consistent state always exists, and the domino effect cannot occur.

The above approaches rely solely on checkpoints, thus the name *checkpoint-based rollback-recovery*. In contrast, *log-based rollback-recovery* uses checkpointing and message logging.[1] Log-based rollback-recovery relies on the assumptions underlied in a *piecewise deterministic (PWD) execution model* [51, 167]. Under the PWD model, each process execution consists of a sequence of deterministic state intervals, each starting with the occurrence of a nondeterministic event. By logging and replaying the nondeterministic events in their exact original order, a process can deterministically

[1]Logging is not confined to messages only. It also includes logging nondeterministic events. Earlier papers in this area have assumed a model in which messages represent nondeterministic events in addition to interprocess communications. In this paper, we use the terms event logging and message logging interchangeably.

recreate its pre-failure state even if it has not been checkpointed. Log-based rollback-recovery in general enables a system to have a recoverable state beyond the most recent set of consistent checkpoints. It is therefore particularly attractive for applications that frequently interact with the *outside world*. The outside world consists of all input and output devices that cannot roll back.

This survey is organized as follows. Section 2 describes the system model, the terminology and the generic issues in rollback-recovery; Section 3 surveys checkpoint-based rollback-recovery protocols, and classifies them into three primary categories: uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing; Section 4 covers log-based recovery techniques including pessimistic logging, optimistic logging and causal logging; Section 5 addresses the implementation issues; Section 6 gives additional references to emerging new research topics and related research areas, and Section 7 concludes the survey. Rollback-recovery techniques that rely on special language constructs such as recovery blocks [144] and transactions [64] are not covered in this survey. Also, we do not address the use of rollback-recovery to tolerate Byzantine failures.

2 Background and Definitions

2.1 System Model and Failure 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 the system. Processes cooperate with each other 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 process executions, and arrows between processes represent messages.

Rollback-recovery protocols generally assume that the communication network is immune to partition, but differ in the assumptions they make about the reliability of interprocess communication. Some protocols assume that the communication subsystem delivers messages reliably in First-In-First-Out (FIFO) order. Other protocols assume that the communication subsystem can lose, duplicate, or reorder messages. The two different assumptions lead to different treatments of in-transit messages, as will be described shortly. Their practical implications are discussed in Section 5.

A process may fail, in which case it loses its volatile state and stops execution according to the fail-stop model [150]. Processes have access to a stable storage device that survives failures. State information saved to the device during failure-free execution then can be used for recovery. The number of tolerated process failures may vary from one to N, and the recovery protocol needs to be accordingly designed. Whether failures that occur during recovery need to be tolerated or not also affects the choice of recovery protocols [51, 157].

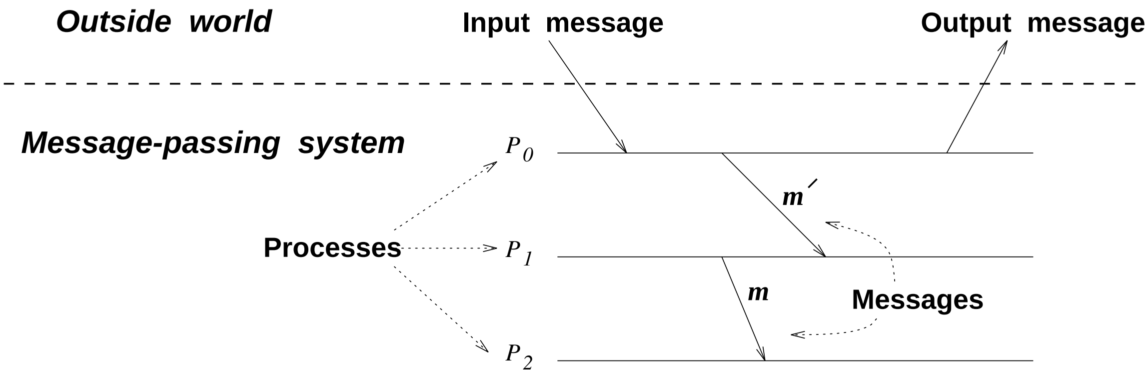

Figure 1: Example message-passing system with three processes.

2.2 Consistent System States

The state of a message-passing system is the collection of the individual states of all participating processes and the states of the communication channels. Intuitively, a consistent system state is one that may occur in a legal execution of a distributed computation. A more precise definition of a *consistent system state* is one in which *every message that has been received is also shown to have been sent in the state of the sender* [38]. For example, the cut in Figure 2(a) straddles a consistent state of the three processes in Figure 1, while the cut in Figure 2(b) straddles an inconsistent cut because process P_2 is shown to have received m but P_1's state does not reflect sending the message.

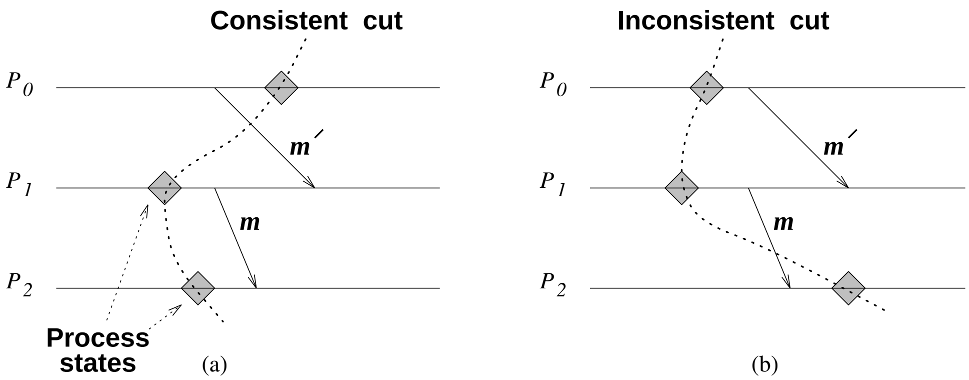

Figure 2: (a) Consistent cut; (b) inconsistent cut.

Messages that are *sent but not yet received* may not cause the system state to be inconsistent. These messages are called *in-transit* messages (see for example message m' with respect to the cut in Figure 2(a)). Whether a consistent system state should include the in-transit messages depends on whether the system model assumes reliable communication channels or not. For reliable communication channels, a consistent state must include in-transit messages because they will always be delivered to their destinations in any legal execution of the system. For example, in Figure 3(a), the reliable communication protocol can handle only the in-transit messages potentially lost in the lossy communication channels during failure-free executions; lost in-transit messages due to process failures need to be separately handled by the rollback-recovery protocol itself. On the other hand, if a system model assumes lossy communication channels, then omitting in-transit messages from the system state does not cause any inconsistency. In such a model, there is no guarantee that the communication subsystem will deliver all messages to their destinations in a legal execution. For example, in Figure 3(b), lost in-transit messages due to rollback-recovery cannot be distinguished from those caused by lossy communication channels; a reliable communication protocol at a higher layer can guarantee the delivery of both types of messages.

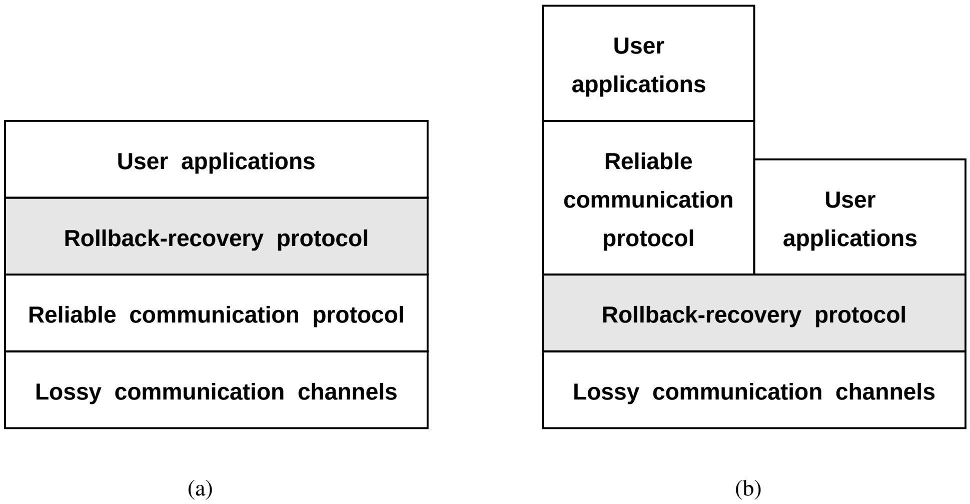

Figure 3: Implementations of rollback-recovery protocols (a) on top of a reliable communication protocol; (b) directly on top of lossy communication channels.

An inconsistent state represents a state that can never occur in any legal execution of the system. Inconsistent states occur because of failures. For example, the inconsistency in Figure 2(b) can occur if process P_1 fails after sending message m to P_2. A fundamental goal of any rollback-recovery protocol is to bring the system into a consistent state when inconsistencies occur due to 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 legal execution.

2.3 Checkpointing Protocols

In checkpointing protocols, each process periodically saves its state on stable storage. The state should contain sufficient information to restart process execution. A *consistent global checkpoint* refers to a set of N local checkpoints, one from each process, which forms a consistent system state. Any consistent global checkpoint can be used for system restoration upon a failure. To minimize the amount of lost work, *the most recent consistent global checkpoint*, called the *recovery line* [144], is the best choice.

Figure 4 gives an example where processes are allowed to take their checkpoints independently, without coordinating with each other. A black bar represents a checkpoint, and each process is assumed to start its execution with an initial checkpoint. Suppose process P_2 fails and rolls back to checkpoint C. The rollback "unsends" message m and so P_1 is required to roll back to checkpoint B to "unreceive" m. The rollback of P_2 thus propagates to P_1, therefore the term *rollback propagation*. P_1's rollback further "unsends" m' and forces P_0 to roll back as well. Such cascading rollback propagation can eventually lead to an unbounded rollback, called the *domino effect* [144], as illustrated in Figure 4. The recovery line for the single failure of P_2 consists of the initial checkpoints. Thus, the system has to roll back to the beginning of its execution and loses all useful work in spite of all the checkpoints that have been taken. To avoid the domino effect, processes need to coordinate their checkpoints so that the recovery line is advanced as new checkpoints are taken.

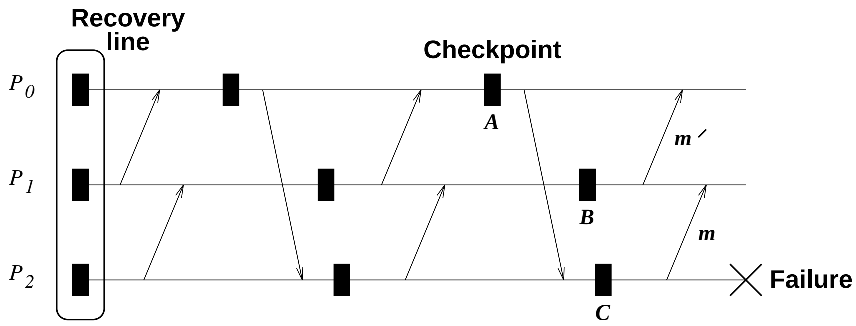

Figure 4: Recovery line, rollback propagation and domino effect.

2.4 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 property is useful when interactions with the outside world are necessary. 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, allowing processes to use uncoordinated checkpointing if desired.[2]

Log-based recovery relies on the assumptions underlied in a *piecewise deterministic (PWD)* execution model [51, 167] and employs an additional logging protocol. Under the PWD assumption, a process execution consists of a sequence of state intervals, each starting with a nondeterministic event such as a message receipt from another process. The execution within each state interval is deterministic. Thus, by logging every nondeterministic event during failure-free execution and replaying the logged events in their original order during recovery, a process can replay its execution beyond the most recent checkpoint. A process state is *recoverable* if there is sufficient information to replay the execution up to that state despite any future failures in the system.

In Figure 5, suppose messages m_5 and m_6 are lost upon the failure affecting both processes P_1 and P_2, while all the other messages survive the failure. Message m_7 becomes an *orphan message* because process P_2 cannot guarantee the regeneration of the same m_6 after the rollback, and P_1 cannot guarantee the regeneration of the same m_7 without the original m_6. As a result, the surviving process P_0 becomes an *orphan process* and is forced to roll back as well. As indicated in Figure 5, process states X, Y and Z then form the *maximum recoverable state* [89], i.e., *the most recent recoverable consistent system state*. Process P_0 (P_2) rolls back to checkpoint A (C) and replays message m_4 (m_2) to reach X (Z). Process P_1 rolls back to checkpoint B and replays m_1 and m_3 in their original order to reach Y.

2.5 Interactions with The Outside World

A message-passing system often interacts with the outside world to receive input data and show the outcome of the computation, or to receive service requests and reply with the requested information. The outside world cannot be relied on to roll back if a failure occurs in the system. For example, a printer cannot roll back the effects of printing a character; an automatic teller machine cannot recover the money that it dispensed to a customer; a deleted file cannot be recovered (unless its state is included as part of the checkpoint [166, 191]). It is therefore necessary to ensure 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. Some rollback-recovery protocols may need to run a special algorithm to ensure the recoverability of the current state, while some protocols can commit output directly without the need for special arrangements.

[2] We use the terms of event logging and message logging interchangeably. Log-based recovery has traditionally been called message logging, as earlier papers have assumed that nondeterministic events can be converted to messages. Also, "message logging" has sometimes been used in the literature to refer to the recording of in-transit messages [42, 187]. This naming convention is not common and we do not use it in this survey.

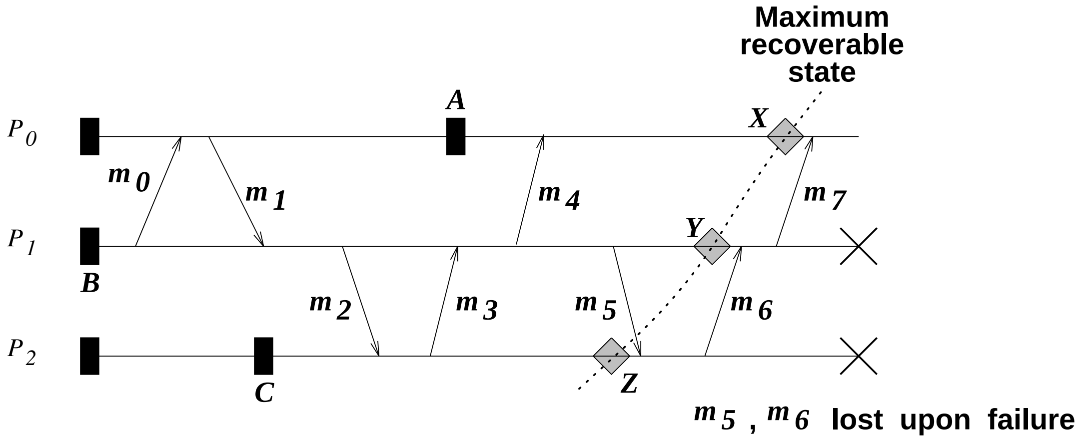

Figure 5: Message logging for deterministic replay.

Similarly, the input messages that a system receives from the outside world may not be reproducible, as it may not be able to regenerate them. Therefore, a recovery protocol must arrange to save the 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.

2.6 Stable Storage

Rollback-recovery uses stable storage to save checkpoints, event logs, and other recovery-related information. Stable storage in rollback-recovery is only an abstraction, although it is often confused with disk storage which is usually used to implement it. Stable storage must ensure that the data stored will persist through the tolerated failure modes. Therefore, in a system that tolerates a single failure, stable storage may consist of the volatile memory of another process [29, 88]. A system that wishes to tolerate an arbitrary number of *transient* failures can implement stable storage by storing information on a reliable disk local to each host. And a system that tolerates non-transient failures must ensure that the recovery information related to a particular process is always stored on a persistent medium outside the host on which the process is running. A highly-available file system can be used in that case [103]. Independent of the technique that implements stable storage, we call an event or a message *fully logged* if it has been stored such that it would persist the tolerated failures in the system.

2.7 Garbage Collection

Checkpoints and event logs consume storage resources. As the application progresses and more recovery information is collected, a subset of the stored information may become useless for recovery. 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 checkpoints, and so all previous checkpoints can be discarded. Garbage collection is an important pragmatic issue in rollback-recovery protocols. Running a special algorithm to discard useless information incurs overhead but may be necessary to free up space on stable storage, posing two conflicting requirements to the system implementors. 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 [144]. It does not rely on piecewise determinism, and so does not need to detect, log, and replay nondeterministic events. Since there is no guarantee that pre-failure execution can be deterministically regenerated after a rollback, it is more suitable for applications that do not frequently interact with the outside world. Checkpoint-based rollback-recovery techniques can be classified into three categories: uncoordinated checkpointing, coordinated checkpointing, and communication-induced checkpointing.

3.1 Uncoordinated Checkpointing

3.1.1 Overview

Uncoordinated (or independent) checkpointing allows each process to decide independently when to take checkpoints. The main advantage is the lower runtime overhead during normal execution because no coordination among processes is necessary. Autonomy in taking checkpoints also allows each process to select appropriate checkpoint positions to further reduce the overhead by saving a smaller amount of state information. The main disadvantage is the possibility of the domino effect, as shown in Figure 4, which may cause a large amount of useful work to be undone regardless of how many checkpoints have been taken. In addition, each process needs to maintain multiple checkpoints, and a garbage collection algorithm needs to be invoked periodically to reclaim the checkpoints that are no longer useful.

During normal execution, the dependencies between checkpoints caused by message exchanges need to be recorded so that a consistent global checkpoint can be determined during recovery. The following direct dependency tracking technique is commonly used in uncoordinated checkpointing [25, 178, 192]. Let $c_{i,x}$ ($0 \leq i \leq N-1$, $x \geq 0$) denote the x^{th} checkpoint of process P_i, where i is called the *process id* and x the *checkpoint index* (we assume each process P_i starts its execution with an initial checkpoint $c_{i,0}$); and let $I_{i,x}$ ($0 \leq i \leq N-1$, $x \geq 1$) denote the *checkpoint interval* (or *interval*) between $c_{i,x-1}$ and $c_{i,x}$. As illustrated in Figure 6, when process P_i at interval $I_{i,x}$ sends a message m to P_j, the pair (i, x) is piggybacked 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 checkpoint $c_{j,y}$ is taken.

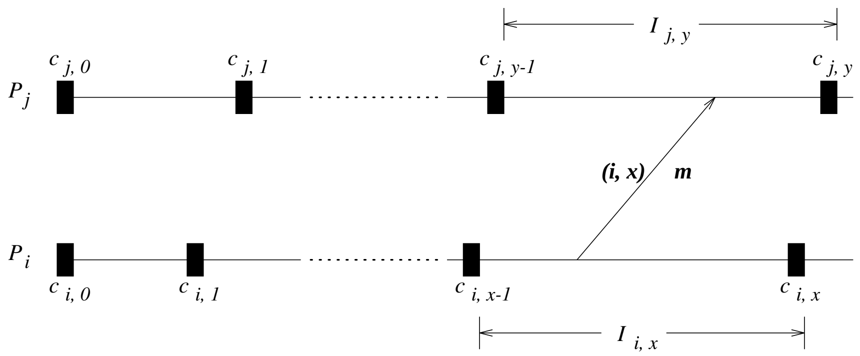

Figure 6: Checkpoint index and checkpoint interval.

If a failure occurs, the rollback initiator will broadcast a dependency_request message to collect all the dependency information maintained separately at each process. When a process receives the dependency_request message, it stops its execution and replies with the stable dependency information and the dependency information associated with its current volatile state (called a *volatile checkpoint*), if available. 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 the rollback_request, if a process's volatile checkpoint belongs to the recovery line, it simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

3.1.2 Dependency Graphs and Recovery Line Calculation

Given the checkpoint and communication pattern shown in Figure 7(a), there are two approaches proposed in the literature to determining the recovery line. The first approach is based on a *rollback-dependency graph* [25, 35, 184] in which each node represents a checkpoint and a directed edge is drawn from $c_{i,x}$ to $c_{j,y}$ if (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 $I_{i,x}$ is rolled back, then $I_{j,y}$ must also be rolled back. The rollback-dependency graph corresponding to the pattern in Figure 7(a) is illustrated in Figure 7(b). To calculate the recovery line, the graph nodes corresponding to the volatile checkpoints of the failed processes P_0 and P_1 are initially marked. A reachability analysis [25, 184] is performed by marking all the nodes reachable from any of the initially marked nodes. The *last unmarked node* of each process then forms the recovery line as shown in Figure 7(b).

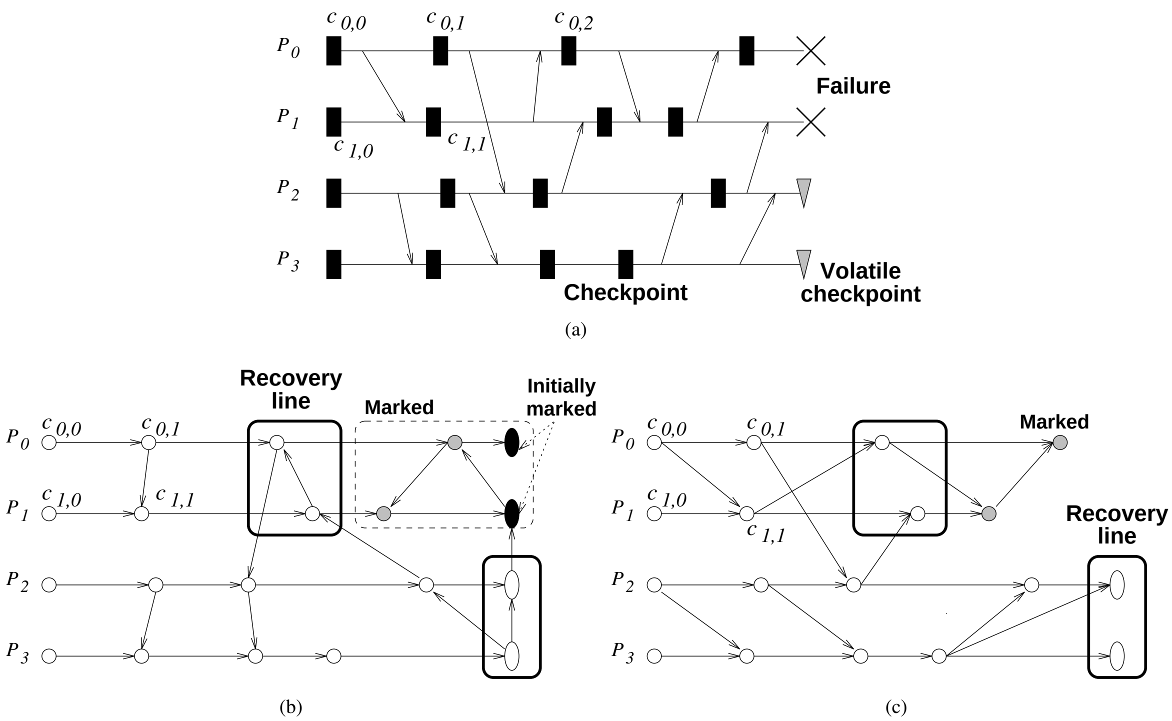

Figure 7: (a) Example checkpoint and communication pattern; (b) rollback-dependency graph; (c) checkpoint graph.

The second approach is based on a *checkpoint graph* [178, 183]. 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 $c_{i,x-1}$ (instead of $c_{i,x}$) to $c_{j,y}$, as shown in Figure 7(c). The recovery line can be calculated by first removing the nodes corresponding to the volatile checkpoints of the failed processes, and then applying the following *rollback propagation algorithm* [178, 187] on the checkpoint graph:

```
/* Initially, all checkpoints are unmarked */

include the last checkpoint of each process in a root set;
mark all the checkpoints strictly reachable from any checkpoint in the root
    set;
while (at least one checkpoint in the root set is marked) {
    replace each marked checkpoint in the root set by the last unmarked
     checkpoint of the same process;
    mark all the checkpoints strictly reachable from any checkpoint in the
     root set;
}
```

the root set is the recovery line.

The example demonstrates that the two approaches are equivalent and result in the same recovery line. The choice usually depends on which graph is more convenient for the issues to be discussed.

3.1.3 Garbage Collection

The garbage collection algorithm for independent checkpointing consists of calculating the recovery line, and discard the obsolete checkpoints before the states that form the line. The calculation proceeds as follows: construct a nonvolatile rollback-dependency graph by omitting the incoming edges of volatile checkpoints (which correspond to volatile dependency information), and initially mark all volatile checkpoints to start the reachability analysis. Figure 8 illustrates the nonvolatile rollback-dependency graph and the global recovery line of Figure 7(a). Only the first checkpoint of each process is obsolete and can be garbage-collected. As demonstrated by the figure, when the global recovery line is unable to advance due to rollback propagation, a large number of nonobsolete checkpoints may need to be retained.

To reduce the number of retained checkpoints, Wang et al. derived the necessary and sufficient condition for a checkpoint to be useful for any future recovery [185, 186]. It was shown that there exists 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 nonvolatile rollback-dependency graph. Figure 9 illustrates the execution of the optimal checkpoint garbage collection algorithm to find these N recovery lines. Since the four nonobsolete checkpoints $\{A, B, C, D\}$ and the four obsolete checkpoints do not belong to the union, they can be safely discarded without affecting the safety of any future recovery. It was also proved that the number of useful checkpoints can never exceed $N(N+1)/2$, and the bound is tight [185].

3.2 Coordinated Checkpointing

3.2.1 Overview

In consistent checkpointing, the processes coordinate their checkpoints to form a global consistent state. Consistent checkpointing is not susceptible to the domino effect, since the processes always restart from the most recent checkpoint. Also, recovery and garbage collection are both simplified, and stable storage overhead is lower than in uncoordinated checkpointing. The main disadvantage is the sacrifice of process autonomy in taking checkpoints. In addition, a coordination session needs to be initiated before committing any output, and checkpoint coordination generally incurs message overhead.

A straightforward approach to coordinated checkpointing is to block interprocess communications until the checkpointing protocol executes [43, 174]. This can be

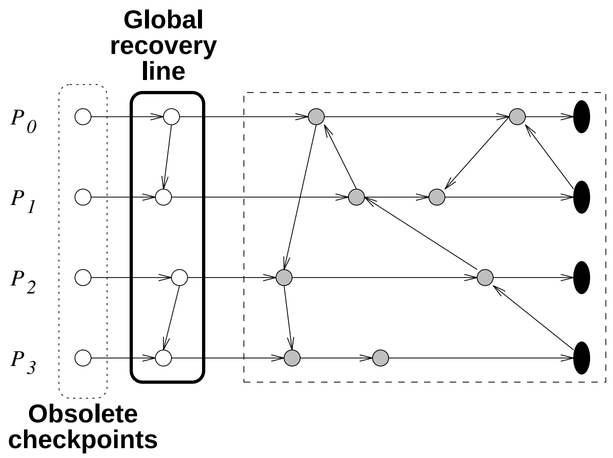

Figure 8: Garbage collection based on global recovery line and obsolete checkpoints.

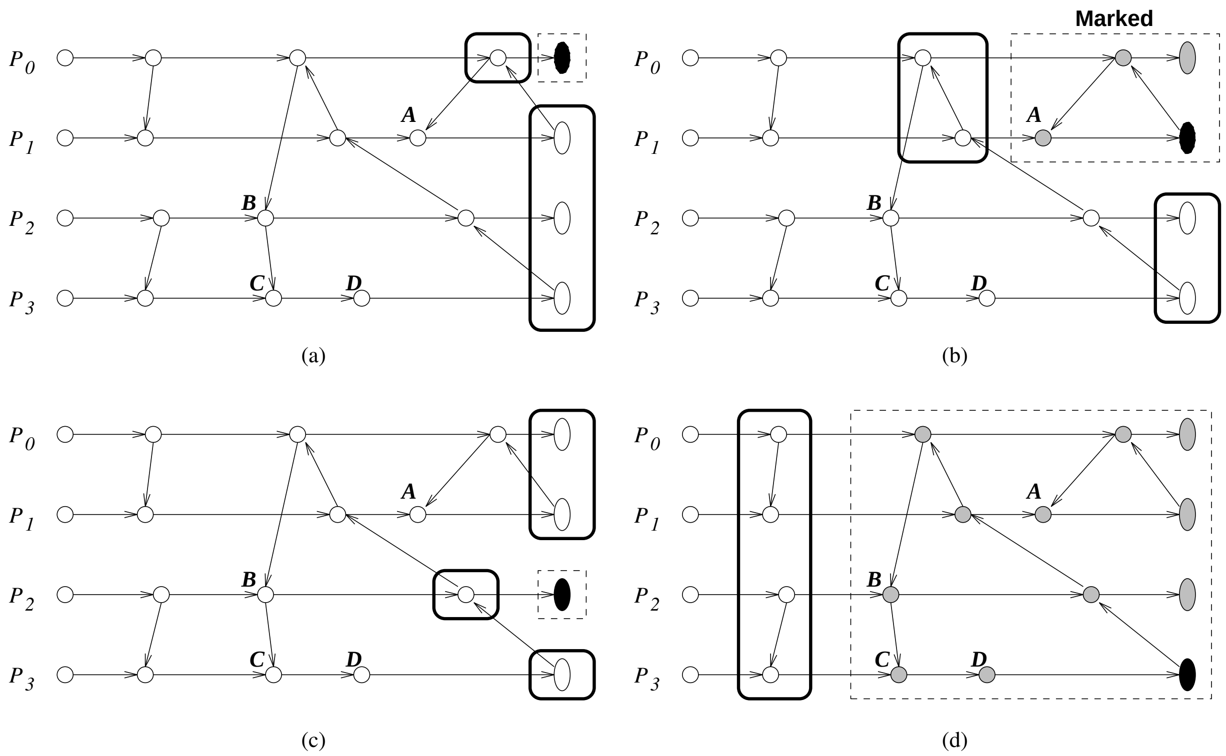

Figure 9: Optimal checkpoint garbage collection.

achieved by using the following two-phase blocking protocol: the initiator (coordinator) broadcasts a checkpoint_request message; when a process receives the checkpoint_request message, it takes a checkpoint, stops sending application messages, and replies with a local_checkpoint_done message; once the initiator receives local_checkpoint_done from every other process, it broadcasts a global_checkpoint_done message; upon receiving global_checkpoint_done, each process commits its new checkpoint and resumes sending application messages. If a failure occurs, a simple recovery procedure is to roll back all processes in the system to the latest committed global checkpoint. When it is desirable to minimize the number of processes involved in a rollback, the general recovery line calculation algorithms based on dependency tracking (as describe in Section 3.1) can still be applied [100].

3.2.2 Nonblocking Checkpoint Coordination

Instead of blocking interprocess communications, an alternative is to shift the responsibility of maintaining checkpoint consistency from the sender side to the receiver side. A fundamental problem in nonblocking checkpoint coordination is to avoid post-checkpoint messages like m in Figure 10(a), which is sent after process P_0 receives checkpoint_request, and received before checkpoint_request reaches P_1. Under the assumption of FIFO channels, this problem can be solved by always generating a checkpoint_request before sending any post-checkpoint messages, and forcing each process to take a checkpoint upon receiving the first checkpoint_request, as illustrated in Figure 10(b). Chandy and Lamport's *distributed snapshot* algorithm [38] provides such a nonblocking checkpoint coordination protocol. (The checkpoint_request message is called a marker in their paper.) Note that, since we only need the checkpoint_request to be processed before any post-checkpoint messages, checkpoint_request can be piggybacked on every post-checkpoint message m and examined by the receiver before m is processed [101], as shown in Figure 10(c). This modification also allows non-FIFO channels. In practice, checkpoint indices can serve as the checkpoint_request messages: a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked checkpoint index [50, 154].

3.2.3 Synchronized Checkpoint Clocks

Loosely synchronous clocks can facilitate checkpoint coordination [42, 143, 177]. More specifically, loosely-synchronized checkpoint clocks can trigger the local checkpointing actions of all participating processes at approximately the same time without the need of broadcasting the checkpoint_request message by a coordinator. A process takes a checkpoint and waits for a period that equals the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process in the system. The process can be assured that all checkpoints belonging to the same coordination session must have been taken without the need of global_checkpoint_done messages. If a failure occurs, it has to be detected within the specified time and the protocol is aborted. To guarantee checkpoint consistency, either the sending of messages is blocked for

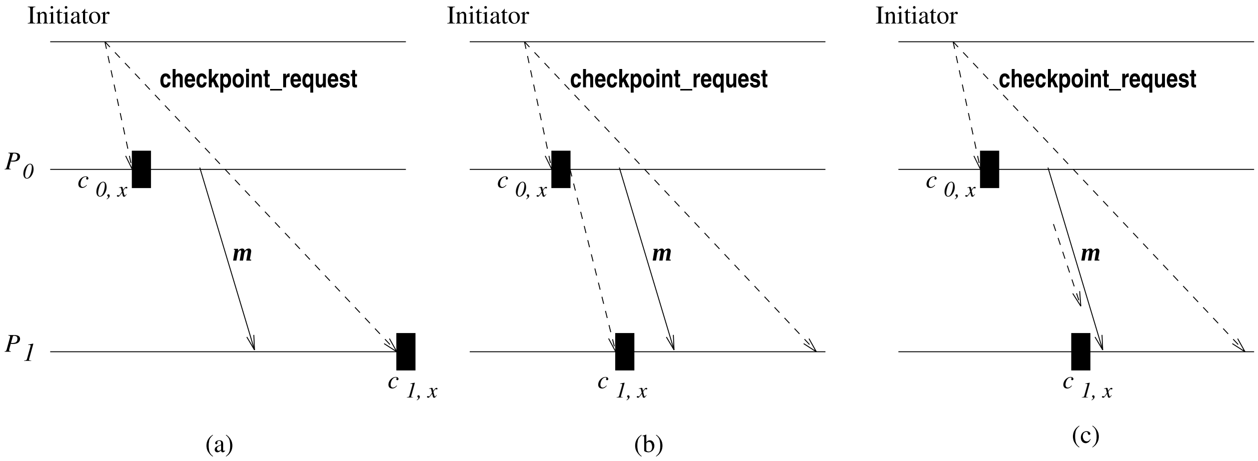

Figure 10: Nonblocking coordinated checkpointing. (a) Checkpoint inconsistency; (b) FIFO channels; (c) non-FIFO channels (short dashed line represents piggybacked checkpoint_request).

the duration of the protocol, or the checkpoint indices can be piggybacked to avoid blocking as explained before.

3.2.4 Minimal Checkpoint Coordination

It is possible to reduce the number of processes involved in a coordinated checkpointing session. Only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints [21, 100]. The following two-phase protocol is due to Koo and Toueg [100]. During the first phase, the checkpoint initiator sends a request to all processes with which it has communicated since the last checkpoint. Upon receiving such request, each process sends a similar message to all processes it has communicated with since the last checkpoint and so on, until all processes are 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 processes that participate. Interprocess communication has to be blocked during this protocol as explained before. In Koo and Toueg's original scheme, if any of the involved processes is not able or not willing to take a checkpoint, then the entire coordination session is aborted; Kim and Park [93] proposed an improved scheme that allows the new checkpoints in some subtrees to be committed while the others are aborted.

3.3 Communication-induced Checkpointing

3.3.1 Overview

Communication-induced checkpointing [81] is another way to avoid the domino effect in uncoordinated checkpointing protocols. A system-wide constraint on the checkpoint and communication pattern is specified to guarantee recovery line progression. Sufficient information is piggybacked on each message so that the receiver can examine the information prior to processing the message. If processing the message would violate the specified constraint, the receiver is forced to take a checkpoint before the processing. In contrast with coordinated checkpointing, no special coordination messages are exchanged. We distinguish two types of communication-induced checkpointing: *model-based checkpointing* maintains certain checkpoint and communication structure that is provably domino effect-free, and *index-based coordination* enforces the consistency between checkpoints with the same index.

3.3.2 Model-based Checkpointing

Several domino effect-free checkpoint and communication models have been proposed in the literature. Russell [147] proved that if within every checkpoint interval all message-receiving events precede all message-sending events, then the system is domino effect-free. Such a model, called an *MRS* 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 [2, 184]. In the *Programmer-Transparent Coordination (PTC)* scheme [98], Kim et al. proved that the domino effect can be eliminated if each process takes an additional checkpoint before processing any message that will cause the process to depend on a checkpoint that it did not previously depend on. Wu and Fuchs [197, 198] proposed that taking a checkpoint immediately after every message-sending event can eliminate rollback propagation and therefore the domino effect. Some heuristics have also been developed to reduce rollback propagation [188, 199], although they in general do not guarantee domino effect-free recovery.

In addition to achieving domino effect-free recovery, another branch of research work aims at providing the benefits of piecewise determinism (such as efficient output commit and recovery) without requiring applications to satisfy the piecewise deterministic model. It is based on the observation that piecewise determinism can be modeled as having a logical checkpoint [91, 179, 190] before every nondeterministic event. Therefore, checkpoint-based rollback recovery can mimic piecewise determinism by taking an actual checkpoint before every nondeterministic event. The main challenge is how to reduce the number of checkpoints while still preserving desirable properties. It has been shown that [182] the three domino effect-free models described in the previous paragraph can all be viewed as special cases of a more general *Fixed-Dependency-After-Send (FDAS)* model: the receiving of any message that causes its receiver P_j to causally depend on a checkpoint $c_{i,x}$ for the first time must precede any sending of messages from the same checkpoint interval. The main advantage of the FDAS model is that *it allows rollback dependency to be tracked on-line*, a property that leads to many desirable features of the piecewise deterministic model. The ability to track rollback dependency is also preserved in the adaptive checkpointing algorithm of Baldoni et al. [16]. In their scheme, an additional boolean vector and another boolean matrix are piggybacked on each message. These data structure allow a receiver to determine if an additional checkpoint needs to be taken to prevent some other checkpoints from becoming useless, i.e., not belonging to any consistent global checkpoints [199].

3.3.3 Index-based Coordination

Checkpoint coordination can also be considered as a mechanism to be incorporated into an uncoordinated checkpointing protocol to eliminate the domino effect. A naive way to employ checkpoint coordination is to start a coordination session whenever a local checkpoint is taken. Alternatively, inconsistency between checkpoints of the same index can be avoided in a lazy fashion if checkpoint index is piggybacked on each message. Upon receiving a message with piggybacked index greater than the local index, the receiver is forced to take a checkpoint before processing the message to avoid inconsistency at the last minute [33, 101].

The lazy coordination protocol described above has two disadvantages. First, the induced checkpoints push the checkpoint indices at some processes higher which may cause more induced checkpoints to be taken and, in the worst case, result in an excessive number of induced checkpoints. Second, the additional checkpoint overhead is determined by the checkpoint and communication pattern and is not otherwise controllable. Wang and Fuchs [189] introduced the notion of *laziness* (a positive integer) to provide a tradeoff between the checkpoint overhead and rollback distance. When a system specifies the laziness to be Z, only checkpoints with the same index which is a multiple of Z are required to be consistent. By increasing the laziness, additional checkpoint overhead can be reduced at the expense of a potentially larger rollback distance. Manivannan and Singhal [119] presented a quasi-synchronous checkpointing algorithm to reduce the number of forced checkpoints. Every process increments its next-to-be-assigned checkpoint index at the same regular time interval to keep the index of the latest checkpoint of each process close to each other. A scheduled checkpoint is skipped if the next-to-be-assigned index is already taken by an induced checkpoint.

4 Log-Based Rollback-Recovery

Log-based rollback-recovery assumes a piecewise deterministic system model in which a process execution consists of a sequence of deterministic state intervals. Each interval starts with the occurrence of a nondeterministic event. Such an event can be the receipt of a message from another process or an internal event to the process. Sending a message, however, is *not* an event in this model. For example, in Figure 5, the execution of process P_0 would be a sequence of four deterministic intervals. The first one starts with the creation of the process, while the remaining three start by the receipt of messages m_0, m_4, and m_7, respectively.

Log-based rollback-recovery protocols save information about the nondeterministic events on stable storage in addition to checkpointing. During recovery, the events in the log are replayed at the same points they occurred during the pre-failure execution. Thus, the failed process reconstructs its pre-failure execution during recovery since the execution within each deterministic interval depends on the nondeterministic event that started it.

Log-based rollback-recovery contrasts checkpointing schemes in one important way. In checkpointing schemes, the system restarts one or more processes after a failure to restore a consistent state. The execution of a failed process during recovery is not necessarily identical to its pre-failure execution. This property simplifies the implementation of failure-recovery but makes it difficult for the system to interact efficiently with the outside world. Log-based rollback-recovery does not have this problem and can interact more efficiently with the outside world.

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 implemented this style of recovery [23, 28]. These systems translated nondeterministic events into messages according to the CSP model [71]. It is important however to emphasize that these protocols are not only limited to message-passing systems. They have found applications in other style of interprocess communication, such as in distributed shared memory systems [37, 170, 197].

Log-based rollback-recovery protocols come in three major variants: pessimistic logging, optimistic logging, and causal logging protocols. They differ in their failure-free performance overhead, latency of output commit, simplicity of recovery and garbage collection, and the potential for rolling back surviving processes.

4.1 Pessimistic Logging

4.1.1 Overview

The basic assumption in pessimistic logging systems is that a failure can occur after every nondeterministic event in the computation. This assumption is "pessimistic" since failures are rare in reality. Pessimistic logging systems arrange for the information about each nondeterministic event to be logged before the event is allowed to affect the computation. For example, a message is not delivered to the application program until it is logged. This form of logging is often called *synchronous logging*. Each process also takes 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 log of events is replayed to recreate the execution. Because the execution is deterministic between nondeterministic events, an exact replay of the pre-failure execution is produced.

Consider the example in Figure 11. During failure-free operation the logs of processes P_0, P_1, and P_2 are $\{m_0, m_4, m_7\}$, $\{m_1, m_3, m_6\}$, and $\{m_2, m_5\}$, respectively. If processes P_1 and P_2 fail as shown, they respectively restart from checkpoints B and C. Each replays its message log and because the execution is deterministic, each restores its pre-failure execution and both will be consistent with the state of P_0 including its receipt of message m_7 from P_1.

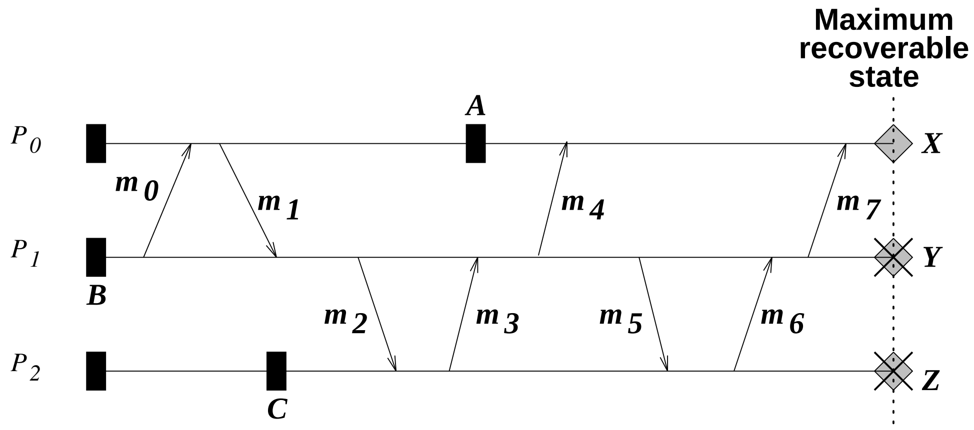

Figure 11: Pessimistic logging.

The state of each process in a pessimistic logging system is always recoverable. This property has four advantages:

- A process can commit output to the outside world without running a special protocol.
- 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. This property is true because a process always recovers to the state that included its most recent interaction with any other process or the outside world.
- 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 taking checkpoints can be determined by trading off the desired runtime performance with the desired protection of the execution.
- There is no need to run a complex garbage collection protocol for the recovery information. Information about nondeterministic events that occurred before the most recent checkpoint and older checkpoints can always be reclaimed since 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.

4.1.2 Techniques for Reducing Performance Overhead

The simplest form of pessimistic logging is to locally save in stable storage information about each event as it occurs and before it affects the application program [72, 73]. This form of logging potentially has a high performance overhead but allows each host to recover independently which is desirable in practical systems [74].

Special hardware that assists logging can lower the overhead. This special hardware can take the form of a fast non-volatile semiconductor memory to implement stable storage [18, 163]. Synchronous logging in such an implementation would be orders of magnitude cheaper than with a traditional implementation of stable storage using magnetic disk devices. Therefore, performance is only slightly affected. Another form of hardware support is to use a special bus that guarantees atomic logging of all messages exchanged in the system [29, 140]. 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 [23, 29].

Some pessimistic logging systems reduce the overhead of synchronous logging without relying on hardware. For example, the sender-based message logging (SBML) protocol logs each message at the sender in volatile memory [88]. A receiver of a message sends an acknowledgment to the sender including the order in which the message is received. The sender includes the receipt order in the log. The log thus contains the information necessary to help the receiver recover from future failures should they occur. This scheme avoids the overhead of accessing stable storage but it can tolerate only one failure and cannot accommodate nondeterministic events internal to a process. Extensions to this technique can tolerate more than one failure in special network topologies [91].

4.1.3 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 [77, 88]. In the example of Figure 11, process P_0 may defer the logging of message m_4 and m_7 until it needs to communicate with another process or the outside world. Thus, these messages 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 decouple logging of an event from its delivery may be susceptible to losing the last messages that were delivered before a failure (an instance of the "last message problem" [124]). This problem occurs only in systems where the communication channels are assumed to be reliable. Consider the example in Figure 11. Assume process P_0 fails after delivering m_4 and m_7 but before logging them. Process P_0 must receive these messages during recovery to be consistent with process P_1. Some protocols that rely on the receiver to log the messages cannot retrieve these messages [77]. This problem does not occur in protocols that rely on sender logging or those that do not assume reliable communication channels [50, 89].

4.2 Optimistic Logging

4.2.1 Overview

Unlike pessimistic logging protocols, optimistic logging protocols [87, 89, 91, 134, 157, 168] log messages *asynchronously*. These protocols make the optimistic assumption that logging will complete before a failure occurs. A volatile log contains information about the events to be logged, and is flushed to stable storage periodically. Optimistic logging does not require the application to block and thus has better failure performance. However, this advantage comes at the expense of more complicated recovery, garbage collection, and output commit compared with pessimistic logging. Should a process fail, the information in the volatile log will be lost and cannot be used during recovery. The execution that depends on the lost information cannot be recovered. Furthermore, if the failed process has sent a message during any of this unrecoverable execution, the receiver of the message then becomes an orphan process and must roll back to "unreceive" this message. For example, suppose P_2 in Figure 12 fails before message m_5 is logged to stable storage. Process P_1 then becomes an orphan process and must roll back to unreceive the orphan message m_6. The rollback of P_1 further forces P_0 to roll back to unreceive m_7. Optimistic logging protocols must therefore perform dependency tracking during failure-free execution. Upon a failure, the dependency tracking information is used to calculate and recover the maximum consistent state of the entire system, in which no process is in an orphan state. The above failure scenario also illustrates that optimistic logging protocols require a nontrivial garbage collection algorithm. While pessimistic logging protocols need only keep the most recent checkpoint of each process, optimistic logging protocols may need to keep additional checkpoints. In the example, process P_1's restart from checkpoint B instead of the most recent checkpoint D due to P_2's failure. Finally, since messages are logged asynchronously, output commit in optimistic logging protocols generally requires multi-host coordination to force the logging progress at some processes 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.

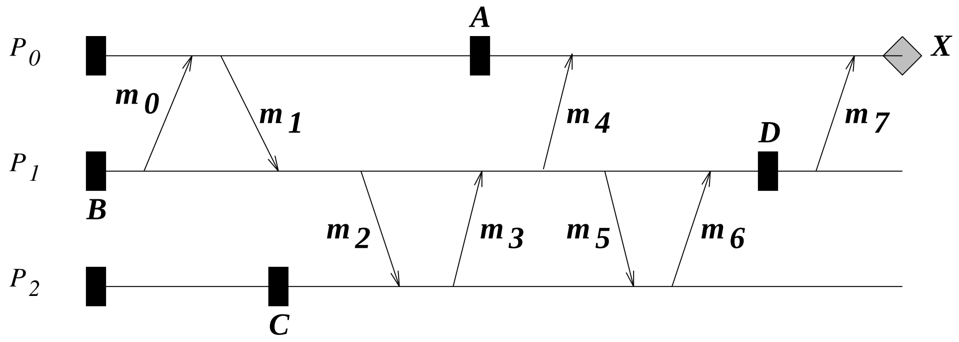

Figure 12: Optimistic logging.

4.2.2 Synchronous vs. Asynchronous Recovery

Recovery in optimistic logging protocols can be either *synchronous* or *asynchronous*. In synchronous recovery [157], 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 its *state interval index* at the beginning of each state interval. Dependency tracking can be either *direct* or *transitive*. In direct dependency tracking [89, 157], the current index of a message sender is piggybacked on each outgoing message to allow the receiver to record the dependency directly caused by the message. These direct dependencies can then be assembled at recovery time to obtain complete dependency information. Alternatively, transitive dependency tracking [157] 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 [87].

In asynchronous recovery, a failed process restarts by sending a rollback announcement broadcast [160] (or recovery message [168]) to start a new *incarnation*. 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 broadcast 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 every other process to correctly detect orphaned states. Strom and Yemini [168] introduced the following blocking at some message receiving events to allow tracking dependency on only *one* incarnation of each process: before process P_i receives 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. To eliminate the blocking and achieve completely asynchronous recovery, the protocol by Smith et al. [160] piggybacks all rollback announcements known to a process on every outgoing message. The protocol was later improved to require piggybacking only a provably minimum amount of information [161].

Another issue in asynchronous recovery protocols is the possibility of *exponential rollbacks*: a single failure in the system may cause a process to roll back an exponential number of times [157]. Figure 13 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, P_1 rolls back to interval $[2, 3]$ and broadcast 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. It was pointed out that Strom and Yemini's original protocol suffers from the exponential rollbacks problem [157]. Three approaches have been proposed to eliminate the problem by ensuring that any process will roll back at most once in response to a single failure. The protocol by Lowry and Strom [117] piggybacks the original rollback announcement from the failed process on every subsequent rollback announcement that it triggers. For example, in Figure 13, process P_1 piggybacks r_0 on r_1. Damani and Garg [45] reduced the number of rollback announcements based on the important observation that *announcing only failures, rather than all rollbacks, suffices to detect orphans*. In other words, rollback announcements generated by non-failed rolled-back processes are always redundant with respect to those generated by failed processes in terms of finding the maximum recoverable state. If rollback announcements are only generated by failed processes, messages like r_1 in Figure 13 no longer exist and so exponential rollbacks will not happen. The recovery protocol by Smith et al. [160, 161] also avoids exponential rollbacks because all rollback announcements are piggybacked on every application message and so always reach a process at the same time.

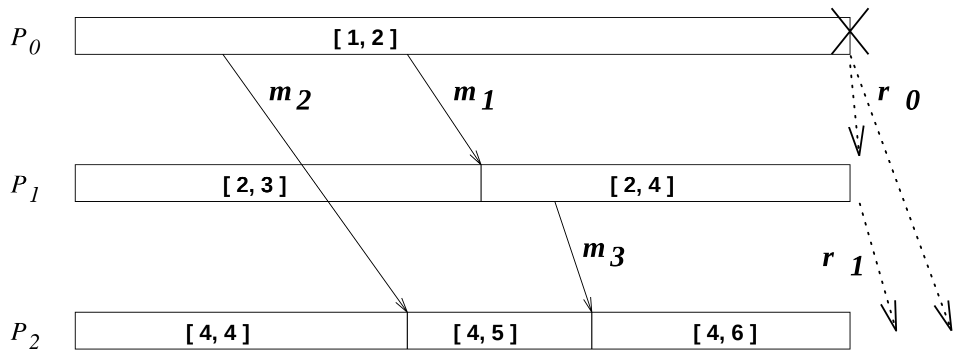

Figure 13: Exponential rollbacks.

4.3 Causal Logging

4.3.1 Overview

Causal logging has the failure-free performance advantages of optimistic logging without making optimistic assumptions. It avoids synchronous access to stable storage except during output commit. Causal logging also retains most of the advantages of pessimistic logging. It allows each process to commit output independently and isolates it 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.

The basic invariant in causal logging is that information about each event that causally precedes the state of a process is either fully logged or is available locally to the process. Consider the example in Figure 14(a). While messages m_5 and m_6 may be lost upon the failure, process P_0 at state X would have information about the nondeterministic events that precede its state in causal order according to Lamport's *happened-before* relation [102]. These events consist of the receipts of messages m_0, m_1, m_2, m_3 and m_4. The information about each of these nondeterministic events is either logged on stable storage or is available locally to process P_0. Thus, process P_0 will be able to *guide* the recovery of P_1 and P_2 because it has the order in which P_1 should replay messages m_1 and m_3 to reach state Y, and the order in which P_2 should replay message m_2 to reach state Z. Such messages can be replayed from the sender log of P_0 or will be regenerated during the recovery of P_1 and P_2.

Each process maintains information about all the events that have causally affected its state. This information acts as an insurance to protect the process 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.

4.3.2 Tracking Causality

The *Manetho* protocol [51] propagates the causal information in an *antecedence graph*. 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. Figure 14(b) shows the antecedence graph of process P_0 of Figure 14(a) at state X. During failure-free operation, each process piggybacks on each application message the receipt orders of its direct and transitive antecedents, ie. its local antecedence graph. The receiver of the message will record 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

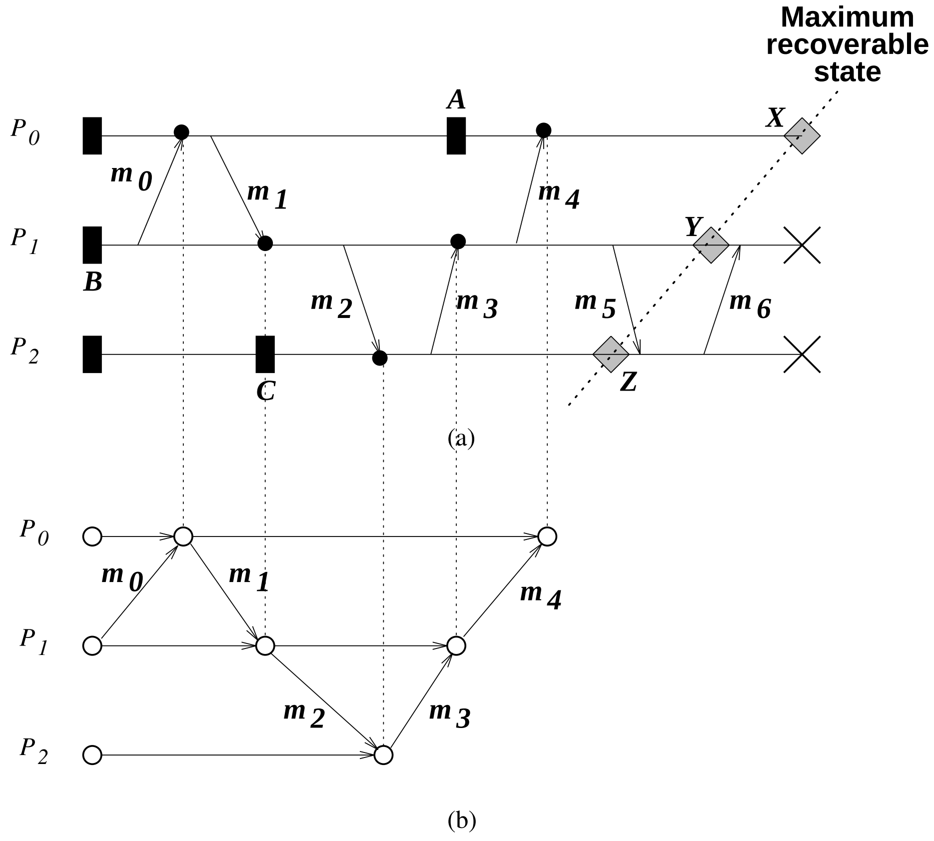

Figure 14: Causal logging. (a) Maximum recoverable states and (b) antecedence graph of P_0 at state X.

on application messages. Thus, any message between two hosts 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 contains 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 [48]. An implementation of this technique shows that it has very low overhead in practice [48].

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 [6, 7]. 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. Sender-based logging is still used to support message replay during recovery. The event information is piggybacked on application messages. However, unlike Manetho, propagation of information about an event stops when it has been recorded in $f + 1$ hosts. For $f < n$, where n is the number of processes, 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 [6]. The Manetho protocol can be considered a member of FBL protocols corresponding to the case of $f = n$.

4.4 Comparison

Various rollback-recovery protocols offer different tradeoffs with respect to performance overhead, latency of output commit, storage overhead, ease of garbage collection, simplicity of recovery, freedom from domino effect, freedom from orphan processes, and the extent of rollback. Table 1 summarizes the comparison between the different variations of rollback-recovery protocols. Uncoordinated checkpointing generally has the lowest failure-free overhead but suffers from potential domino effect. This can be avoided by paying certain degree of performance overhead either to coordinate checkpoints or to log messages under the assumption of piecewise determinism. The PWD assumption also has the additional advantages of allowing faster output commits and orphan-free recovery. 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 may need 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 [186]. Several checkpoints may need to be kept under optimistic logging, depending on the logging progress.

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 indicate that the main difficulty in implementing these protocols lies in the complexity of handling recovery [48]. It is interesting that all commercial implementations of message

	Uncoordinated Checkpointing	Coordinated Checkpointing	Pessimistic Logging	Optimistic Logging	Causal Logging
PWD Assumed?	No	No	Yes	Yes	Yes
Overhead	Low	Higher	Highest	Higher	Higher
Output Commit	Not possible	Very slow	Fastest	Slow	Fast
Checkpoint/process	Several	1	1	Several	1
Garbage Collection	Complex	Simple	Simple	Complex	Complex
Recovery	Complex	Simple	Simple	Complex	Complex
Domino Effect	Possible	Not possible	Not possible	Not possible	Not possible
Orphans	Possible	Possible	Not possible	Possible	Not possible
Rollback Extent	Unbounded	Last checkpoint	Last checkpoint	Some previous checkpoint	Last checkpoint

Table 1: Comparison between different flavors of rollback-recovery protocols.

logging use pessimistic logging because it simplifies recovery [29, 74].

Several recent studies have also challenged some premises which many recovery protocols rely on. Many of these protocols have been incepted in the 1980's. During that era, processor speed and network bandwidth were such that communication overhead was deemed too high, especially when compared to the cost of stable storage access [26]. In such platforms, a protocol that requires multi-host coordination incurs a large overhead due to the necessary control messages that carry out the protocol. A protocol that does not require such communication overhead at the expense of more stable storage access would perform 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.[3] This change in the equation suggests a fresh look at the premises of many rollback-recovery protocols. Specifically, recent results have shown that [53, 106, 135]:

- Stable storage access is now the major source of overhead in checkpointing systems. Communication overhead is much lower in comparison. Such changes favor coordinated checkpointing schemes over message logging or independent 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 [53]. Message logging systems can implement efficient protocols for committing output and logging input that are not possible in checkpoint-only systems.

[3] 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.

- 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 issues in some detail.

5.2 Checkpointing

All available studies have shown that writing the state of a process to stable storage is an important contributor to the performance overhead [135]. The simplest way to save the state of a process is to suspend it, save its address space on stable storage, and then resume it [92, 99, 106, 114, 159, 194]. 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 Reducing Checkpointing Overhead

Concurrent checkpointing techniques greatly reduce the overhead of saving the state of a process [109–111]. Concurrent checkpointing does not suspend the execution of the process while the checkpoint is saved on stable storage. It relies on the memory protection hardware that is commonly available in modern computer architectures. 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 will incur 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.

Adding incremental checkpointing to concurrent checkpointing can further reduce the overhead [50]. 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 [12]. A public domain package implementing these techniques is available [136].

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 [137]. 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 [137].

5.2.2 System-level versus User-level Implementations

Support for checkpointing can be implemented in the kernel [48, 86, 135], or it can be implemented by a library linked with the user program [62, 106, 136, 159, 165, 191].

Kernel-level implementations are more powerful because they can also capture kernel data structures that support the checkpointed 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 [62]. 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 [149, 191].

5.2.3 Compiler Support

A compiler can be instrumented to generate code that supports checkpointing [108]. 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 checkpointed, therefore avoiding saving 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 several 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 could provide hints to the compiler about where checkpoints could be inserted or what data variables should be stored [24, 138, 152]. The compiler may also be trained by running the application in an iterative manner and observing its behavior [108]. 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 [9]. The execution point after each major garbage collection provides a convenient place to take a checkpoint at a minimum cost.

5.2.4 Coordinated versus Uncoordinated Checkpointing

Many checkpointing protocols were incepted at a time where the communication overhead far exceeded the overhead of accessing stable storage [26]. 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 [50, 125]. 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.

5.3 Communication Protocols

Rollback-recovery complicates the implementation of protocols used for interprocess communications. Some protocols offer the abstraction of reliable communication channels such as connection-based protocols like TCP [139] or RPC-style communications [27]. Alternatively, other protocols offer the abstraction of an unreliable datagram service such as UDP [139]. Each type of abstraction requires additional support to operate properly across failures and recoveries.

5.3.1 Location-Independent Identities and Redirection

For all communication protocols, a rollback-recovery system must mask the actual identity and location of a process or a remote port from the application program. This masking is necessary to prevent any application program from acquiring a dependency on the location of a certain process. Such a dependency would make it impossible to restart a process on a different machine after a failure. A solution to this problem is to assign a location-independent, logical identifier to each process in the system [176]. The system translates the logical identifier to the actual network address of the process in an application-transparent manner. This scheme also allows the system to appropriately redirect communication to a restarting process after a failure.

5.3.2 Reliable Channel Protocols

Identity masking and communication redirection after a failure 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 due to 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.

5.4 Message Logging

Message logging introduces two sources of overhead. First, each message must in general be copied in the local memory of the process. Second, the volatile log must be flushed on stable storage. 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 interprocess 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 message logging with the implementation of the communication protocol and share the message log with the transmission buffers. This scheme would avoid the extra copying of the message. Logging at the receiver is more expensive because it is in the critical path and no such sharing between the message logging and the communication protocol logic can be implemented.

Another optimization for sender-based logging systems is to use copy-on-write to avoid making extra-copying. 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 [53].

5.4.1 Message Logging and Coordinated Checkpointing

Message logging has been traditionally presented as a scheme that allows the system to use uncoordinated checkpointing with no domino effect. However, there is nothing that prevents the system from using coordinated checkpointing in a message logging system [53]. Such a scheme has many advantages with respect to performance and simplicity. It retains the ability to perform fast output commit as in log-based systems. It also retains the simplicity of recovery and garbage collection that comes from coordinated checkpointing. Furthermore, it allows a sender-based logging system to avoid flushing the logs on stable storage, reducing the overhead and complexity of maintaining logs on stable storage. The combination of coordinated checkpointing and message logging has been shown to outperform one with uncoordinated checkpointing and message logging [53]. Therefore, the purpose of logging should no longer be the avoidance of taking uncoordinated checkpointing but the desire for enabling fast output commit.

5.5 Stable Storage

Magnetic disks have been the medium of choice for implementing stable storage. Although 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 and reliability needed to implement stable storage [48]. The KitLog package offers a log abstraction on top of which support for checkpointing and message logging can be implemented. The package runs in conventional UNIX systems and bypasses the UNIX file system by accessing the disk in raw mode [146].

There have been also several attempts at implementing stable storage using non-volatile semiconductor memory [18]. 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.

5.6 Support for Nondeterminism

Nondeterminism occurs when the application program interacts with the operating system through system calls and upcalls. Log-based systems must track the nondeterminism during failure-free operation and replays it with the same effect during recovery.

5.6.1 System Calls

System calls in general can be classified into three types. 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. Therefore, the previous result is simply returned to the application. 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 reexecution can be very complex [48, 149].

5.6.2 Asynchronous signals

Different flavors of logging have been suggested with different performance and resilience characteristics [7]. These protocols, however, do not support general forms of nondeterminism that are found in practice. It is inefficient for example to track the nondeterminism resulting from software interrupts such as UNIX signals. Such signals must be applied at the same execution points during replay to reproduce the same result. Systems that support this form of nondeterminism simply take a checkpoint after the occurrence of each signal, which can be very expensive [48]. Alternatively, the system may convert these asynchronous signals to synchronous messages such as in Targon/32 [29], or it may queue the signals until the application polls for them such as in Delta-4 [22, 39]. 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 [62]. The primitives are instrumented to insert an entry in the log identifying the calling thread and the nature of the synchronization operation [62]. 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 (due to a programmer's error, for instance), execution replay may not be possible.

A promising technique for solving this problem is to use instruction counters to efficiently track nondeterminism due to asynchronous software interrupts and multi-threading on single-processor systems. An instruction counter is a register that is decremented upon the execution of each instruction. The hardware generates an exception when the register content becomes 0. An Instruction counter can 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, an exception is generated and propagated to 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. The use of instruction counters has been suggested for debugging shared memory parallel programs [36, 122, 148].

Instruction counters can be used in rollback-recovery to track the number of instructions that occur between asynchronous interrupts. A replay system can use the instruction count to force the execution of the same number of instructions between asynchronous interrupts. An instruction counter can be implemented in hardware, such as in the PA-RISC precision architecture. It also can be emulated in software [122]. A recent implementation on a DEC 3000/400 workstation shows that the overhead of program instrumentation and tracking nondeterminism is less than 6% for a variety of user programs and synthetic benchmarks [158].

5.7 Dependency Tracking

There are three forms for implementing dependency tracking. The first is the simplest and consists of tagging the message with an index or a sequence number [86]. Dependency tracking also can take the form of piggybacking a vector or a graph on top of each message. There are techniques for optimizing these forms of tracking by exploiting the semantics of the communication system and by piggybacking only incremental changes over application messages. Prototype implementations have shown that the overhead resulting from tracking is negligible compared to the overhead of checkpointing or logging [48].

5.8 Recovery

Handling execution restart and replay is a difficult part of implementing a rollback-recovery system [48, 104]. 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 [48]. This can be done for instance by intercepting system calls that return environment-specific results and replace these results with abstract values under the control of the recovery system [149]. 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. In any case, the system must reconstruct the state of the process and also the supporting kernel-level data structures during recovery.

6 Related Work

Most existing papers on rollback-recovery either assume all processes are piecewise deterministic or do not take advantage of piecewise determinism at all. In practice, it is important to support systems consisting of both deterministic and nondeterministic processes [87, 90]. One challenge is to handle unreplayable nondeterministic events while still preserving the advantages of piecewise determinism [41, 184, 190]. Although most rollback-recovery techniques were originally designed for tolerating hardware failures, they have also been applied to software and protocol error recovery [169, 184, 190, 193]. Rollback-recovery in shared-memory and distributed shared-memory systems has also been extensively studied [4, 20, 54, 75, 80–83, 109, 132, 170, 197, 198].

This survey has covered mostly rollback-recovery techniques which do not require or take advantage of special linguistic supports. A substantial amount of research efforts has also focused on coordinated recovery based on special language constructs such as recovery blocks and conversations [34, 65, 66, 79, 94, 96, 144, 145, 200]. Nett et al. addressed recovery problems in dynamic action models [126–128]. Kim et al. addressed recovery problems in the Programmer-Transparent Coordination (PTC) scheme [95, 97, 98]. Orphan elimination problem in nested transaction systems has also been studied [69, 70, 113].

Theoretical aspects of distributed snapshots also have been studied outside the context of recovery [1, 5, 38, 43, 67, 101, 162, 180]. Several fundamental properties regarding consistent global states have been derived [13, 16, 120, 131, 184]. Vector timestamps [55, 121, 151, 155] and the *context graph* used in em Psync [133] bear similarities to the various dependency tracking techniques. Checkpointing and message logging can also be used to facilitate the debugging of parallel and distributed programs [57, 63, 129, 130]. In the area of distributed discrete-event simulation [59, 124], the Time Warp optimistic approach, which inspired the seminal work on optimistic message logging [168], uses rollbacks to cancel erroneous computations due to the out-of-order arrivals of time-stamped event messages [59, 60, 85, 118, 141].

7 Conclusions

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. Uncoordinated checkpointing generally has the least constraints and the lowest overhead. But since it suffers from potential domino effect, uncoordinated checkpointing often needs to be combined with other techniques to be useful in practice. For applications involving multiple processes executing in coordinated steps, coordinated checkpointing is often the natural choice to simplify both failure-free and recovery-time operations. It can also be combined with log-based recovery protocols to simplify the garbage collection task. When desirable, communication-induced checkpointing with index-based coordination can be used to coordinate checkpoints in a distributed fashion. For applications that frequently interact with the outside world, log-based rollback recovery based on piecewise determinism is often a better choice because it allows efficient output commit. The simplicity of pessimistic logging makes it attractive for practical applications which can tolerate a higher failure-free overhead. 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 provides a tradeoff between the overhead of logging and the extent of rollback upon a failure. Finally, model-based checkpointing can be used to mimic piecewise determinism by taking additional checkpoints instead of relying on message logging.

Acknowledgement

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

References

[1] A. Acharya and B. R. Badrinath. Recording distributed snapshots based on causal order of message delivery. *Information Processing Letters*, 44(6), December 1992.

[2] A. Acharya and B. R. Badrinath. Checkpointing distributed applications on mobile computers. In *Proc. the Third International Conference on Parallel and Distributed Information Systems*, pages 73–80, September 1994.

[3] M. Ahamad and L. Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 2–11, 1989.

[4] R. E. Ahmed, R. C. Frazier, and P. N. Marinos. Cache-aided rollback error recovery (carer) algorithms for shared-memory multiprocessor systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 82–88, 1990.

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

[6] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 145–154, 1993.

[7] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 229–236, May 1995.

[8] L. Alvisi and K. Marzullo. Trade-offs in implementing causal message logging protocols. In *ACM Annual Symp. on the Priciples of Distributed Computing*, pages 58–67, May 1996.

[9] A. W. Appel. A runtime system. Technical Report CS-TR-220-89, Department of Computer Science, Princeton University, 1989.

[10] A. Arora and M. Gouda. Distributed reset. *IEEE Trans. Comput.*, 43(9):1026–1038, September 1994.

[11] O. Babaoglu. Fault-tolerant computing based on Mach. In *Proceedings of the Usenix Mach Workshop*, pages 186–199, October 1990.

[12] 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*, pages 78–86, 1981.

[13] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems, Ed. S. Mullender*, pages 55–96. Addison-Wesley, 1993.

[14] D. F. Bacon. File system measurements and their application to the design of efficient operation logging algorithms. *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 21–30, 1991.

[15] D. F. Bacon. Transparent recovery in distributed systems. *ACM Oper. Syst. Review*, pages 91–94, April 1991.

[16] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. Consistent checkpointing in message passing distributed systems. Technical Report No. 2564, INRIA, France, June 1995.

[17] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. On modeling consistent checkpoints and the domino effect in distributed systems. Technical Report No. 2569, INRIA, France, June 1995.

[18] 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*, pages 285–293, February 1988.

[19] J. P. Banâtre, M. Banâtre, and G. Muller. Architecture of fault-tolerant multiprocessor workstations. In *Workshop on Workstation Operating Systems*, pages 20–24, 1989.

[20] M. Banatre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin. An architecture for tolerating processor failures in shared-memory multiprocessors. Technical Report 707, IRISA, Rennes, France, March 1993.

[21] G. Barigazzi and L. Strigini. Application-transparent setting of recovery points. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 48–55, 1983.

[22] P.A. Barrett, A.M. Hilborne, P. Verissimo, L. Rodrigues, P.G. Bond, D.T. Seaton, and N.A. Speirs. The Delta-4 extra performance architecture XPA. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 481–488, June 1990.

[23] J. F. Bartlett. A NonStop Kernel. In *Proc. 8th ACM Symp. on Operating Systems Principles*, pages 22–29, 1981.

[24] M. Beck, J. S. Plank, and G. Kingsley. Compiler-assisted checkpointing. Technical Report CS-94-269, University of Tennessee at Knoxville, December 1994.

[25] B. Bhargava and S. R. Lian. Independent checkpointing and concurrent rollback for recovery - An optimistic approach. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 3–12, 1988.

[26] B. Bhargava, S.-R. Lian, and P.-J. Leu. Experimental evaluation of concurrent checkpointing and rollback recovery algorithms. In *Proc. Int. Conf. Data Eng.*, pages 182–189, March 1990.

[27] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[28] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault-tolerance. In *Proc. 9th ACM Symp. on Operating Systems Principles*, pages 90–99, October 1983.

[29] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Trans. Comput. Syst.*, 7(1):1–24, February 1989.

[30] N. S. Bowen and D. K. Pradhan. Survey of checkpoint and rollbak recovery techniques. Technical Report TR-91-CSE-17, Dept. of Electrical and Computer Engineering, University of Massachusetts, Amherst, July 1991.

[31] N. S. Bowen and D. K. Pradhan. Virtual checkpoints: Architecture and performance. *IEEE Trans. Comput.*, 41(5):516–525, May 1992.

[32] N. S. Bowen and D. K. Pradhan. Processor- and memory-based checkpoint and rollback recovery. *IEEE Computer Magazine*, pages 22–31, February 1993.

[33] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proc. IEEE 4th Symp. on Reliability in Distributed Software and Database Systems*, pages 207–215, 1984.

[34] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Trans. Software Eng.*, SE-12(8):811–826, 1986.

[35] J. Cao and K. C. Wang. An abstract model of rollback recovery control in distributed systems. *ACM Oper. Syst. Review*, pages 62–76, October 1992.

[36] T. Cargill and B. Locanthi. Cheap hardware support for software debugging and profiling. *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 82–83, October 1987.

[37] J. 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*, pages 515–523, 1993.

[38] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[39] M. Chérèque, D. Powell, P. Reynier, J-L. Richier, and J. Voiron. Active replication in Delta-4. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 28–37, July 1992.

[40] G.-M. Chiu and C.-R. Young. Efficient rollback-recovery technique in distributed computing systems. *IEEE Trans. Parallel and Distributed Syst.*, 7(6):565–577, June 1996.

[41] E. Cohen, Y. M. Wang, and G. Suri. When piecewise determinism is almost true. In *Proc. Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 66–71, December 1995.

[42] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 12–20, 1991.

[43] C. Critchlow and K. Taylor. The inhibition spectrum and the achievement of causal consistency. Technical Report TR 90-1101, Cornell University, February 1990.

[44] 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*, pages 424–433, June 1994.

[45] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 108–115, 1996.

[46] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of checkpointing and rollback techniques. Technical Report O3.1.8 and O3.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium, June 1993.

[47] G. Deconinck, J. Vounckx, R. Lauwereins, and J. A. Peperstraete. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. In *Proc. IASTED Int. Conf. on Modelling and Simulation*, pages 262–265, May 1993.

[48] E. N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. PhD thesis, Department of Computer Science, Rice University, October 1993. Also available as Technical Report TR93-212.

[49] E. N. Elnozahy. *Fault Tolerance for Clusters of Workstations, M. Banatre and P. Lee (Editors)*, chapter 8. Springer Verlag, August 1994.

[50] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 39–47, October 1992.

[51] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[52] 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*, pages 18–27, July 1992.

[53] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 298–307, 1994.

[54] M. J. Feeley, J. S. Chase, V. Narasayya, and H. M. Levy. Integrating coherency and recovery in distributed systems. In *Proc. Symp. on Operating System Design and Implementation*, 1994.

[55] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. 11th Australian Computer Science Conference*, pages 55–66, February 1988.

[56] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global states of a distributed system. *IEEE Trans. Software Eng.*, SE-8(3):198–202, May 1982.

[57] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 134–141, 1990.

[58] T. M. Frazier and Y. Tamir. Application-transparent error-recovery techniques for multicomputers. In *The Fourth Conferences on Hypercubes, Concurrent Computers, and Applications*, pages 103–108, March 1989.

[59] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[60] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Proc. SCS Multiconference on Distributed Simulation*, pages 61–67, July 1988.

[61] V. K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, November 1992.

[62] A. P. Goldberg, A. Gopal, K. Li, R. E. Strom, and D. F. Bacon. Transparent recovery of Mach applications. In *First USENIX Mach Workshop*, October 1990.

[63] A. P. Goldberg, A. Gopal, A. Lowry, and R. E. Strom. Restoring consistent global states of distributed computations. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[64] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[65] S. T. Gregory and J. C. Knight. A new linguistic approach to backward error recovery. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 404–409, 1985.

[66] S. T. Gregory and J. C. Knight. On the provision of backward error recovery in production programming languages. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 506–511, 1989.

[67] B. Groselj. Bounded and minimum global snapshots. *IEEE Parallel and Distributed Technology*, 1(4), November 1993.

[68] V. Hadzilacos. An algorithm for minimizing roll back cost. In *Proc. ACM Symp. on Principles of Database Systems*, pages 93–97, 1982.

[69] M. Herlihy, N. Lynch, M. Merritt, and W. Weihl. On the correctness of orphan management algorithms. *J. of ACM*, 39(4):881–930, October 1992.

[70] M. Herlihy and M. McKendry. Timestamp-based orphan elimination. *IEEE Trans. Software Eng.*, 15(7):825–831, July 1989.

[71] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[72] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 2–9, June 1993.

[73] Y. Huang and C. Kintala. A software fault tolerance platform. In *Practical Reusable Software, Ed. B. Krishnamurthy*, pages 223–245. John Wiley & Sons, 1995.

[74] Y. Huang and Y. M. Wang. Why optimistic message logging has not been used in telecommunications systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 459–463, June 1995.

[75] G. G. Richard III and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 58–67, 1993.

[76] S. Israel and D. Morris. A non-intrusive checkpointing protocol. In *The Phoenix Conference on Communications and Computers*, pages 413–421, 1989.

[77] P. Jalote. Fault tolerant processes. *Distributed Computing*, 3:187–195, 1989.

[78] P. Jalote. *Fault Tolerance in Distributed Systems*. Englewood Cliffs, New Jersey:Prentice-Hall, 1994.

[79] P. Jalote and R. H. Campbell. Atomic actions for fault-tolerance using CSP. *IEEE Trans. Software Eng.*, SE-12(1):59–68, 1986.

[80] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 42–51, October 1994.

[81] B. Janssens and W. K. Fuchs. Experimental evaluation of multiprocessor cache-based error recovery. In *Proc. Int. Conf. Parallel Processing*, pages I–505–I–508, 1991.

[82] B. Janssens and W. K. Fuchs. Relaxing consistency in recoverable distributed shared memory. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 155–163, June 1993.

[83] B. Janssens and W. K. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 34–41, October 1994.

[84] D. P. Jasper. A discussion of checkpoint restart. *Software Age*, October 1969.

[85] D. R. Jefferson. Virtual time. *Trans. on Programming Languages and Systems*, 7(3):404–425, July 1985.

[86] D. B. Johnson. *Distributed system fault tolerance using message logging and checkpointing*. PhD thesis, Department of Computer Science, Rice University, December 1989.

[87] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 86–95, October 1993.

[88] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 14–19, 1987.

[89] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms*, 11:462–491, 1990.

[90] D. B. Johnson and W. Zwaenepoel. Transparent optimistic rollback recovery. *ACM Oper. Syst. Review*, pages 99–102, April 1991.

[91] T. T-Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 454–461, 1991.

[92] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. Technical Report IR-258, Vrije Universiteit, Amsterdam, October 1991.

[93] J. L. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Trans. Parallel and Distributed Syst.*, 4(8):955–960, August 1993.

[94] K. H. Kim. Approaches to mechanization of the conversation scheme based on monitors. *IEEE Trans. Software Eng.*, SE-8(3):189–197, May 1982.

[95] K. H. Kim. Programmer-transparent coordination of recovering concurrent processes: Philosophy and rules for efficient implementation. *IEEE Trans. Software Eng.*, 14(6):810–821, June 1988.

[96] K. H. Kim. The distributed recovery block scheme. In *Software Fault Tolerance, Ed. M. R. Lyu*, pages 189–209. John Wiley & Sons, 1995.

[97] K. H. Kim and J. H. You. A highly decentralized implementation model for the Programmer-Transparent Coordination (PTC) scheme for cooperative recovery. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 282–289, 1990.

[98] K. H. Kim, J. H. You, and A. Abouelnaga. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 130–135, 1986.

[99] B. A. Kingsbury and J. T. Kline. Job and process recovery in a UNIX-based operating system. In *Usenix Association, Winter Conference Proceedings, 1989*, pages 355–364, January 1989.

[100] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng.*, SE-13(1):23–31, January 1987.

[101] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

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

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

[104] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[105] P. A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*. Wien: Springer-Verlag, 1990.

[106] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Department of Computer Science, Carnegie Mellon University, February 1993.

[107] H. V. Leong and D. Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 227–234, 1994.

[108] C. C. Li and W. K. Fuchs. CATCH: Compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 74–81, 1990.

[109] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpointing for parallel programs. In *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[110] K. Li, J. F. Naughton, and J. S. Plank. Checkpointing multicomputer applications. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 2–11, 1991.

[111] K. Li, J. F. Naughton, and J. S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *Int. J. of Parallel Program.*, 20(3):159–180, June 1992.

[112] L. Lin and M. Ahamad. Checkpointing and rollback-recovery in distributed object based systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 97–104, 1990.

[113] B. Liskov, R. Scheifler, E. Walker, and W. Weihl. Orphan detection. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 2–7, 1987.

[114] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter 1992 Technical Conference*, pages 283–290, January 1992.

[115] 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*, pages 58–65, July 1992.

[116] A. Lowry, J. R. Russell, and A. P. Goldberg. Optimistic failure recovery for very large networks. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 66–75, 1991.

[117] A. Lowry and R. E. Strom. Some problems with optimistic recovery and their solutions. Personal communications, December 1992.

[118] V. Madisetti, J. Walrand, and D. Messerschmitt. WOLF: A rollback algorithm for optimistic distributed simulation systems. In *Simulation Conference Proceedings*, pages 296–305, December 1988.

[119] D. Manivannan and M. Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 100–107, 1996.

[120] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. Tech. Rep. No. OSU-CISRC-5/96-TR33, Dept. of Computer and Information Science, Ohio State University, 1996.

[121] F. Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, October 1988.

[122] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989.

[123] P. M. Merlin and B. Randell. State restoration in distributed systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 129–134, June 1978.

[124] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[125] G. Muller, M. Hue, and N. Peyrouz. Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment. *Lecture Notes in Computer Science: Dependable Computing- EDCC-1*, pages 491–508, October 1994.

[126] E. Nett. The recovery problem in distributed systems. In *Proc. Workshop on the Future Trends of Distributed Computing Systems in the 1990's*, pages 357–365, 1988.

[127] E. Nett, R. Kroger, and J. Kaiser. Implementing a general error recovery mechanism in a distributed operating system. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 124–129, 1986.

[128] E. Nett and B. Weiler. Nested dynamic actions - How to solve the fault containment problem in a cooperative action model. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 106–115, 1994.

[129] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proc. Supercomputing '92*, pages 502–511, November 1992.

[130] R. H. B. Netzer and J. Xu. Adaptive message logging for incremental program replay. *IEEE Parallel and Distributed Technology*, 1(4):32–39, November 1993.

[131] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel and Distributed Syst.*, 6(2):165–169, February 1995.

[132] N. Neves, M. Castro, and P. Guedes. A checkpointing protocol for an entry consistent shared memory system. In *Proc. 13th ACM Symp. on Principles of Distr. Computing*, 1994.

[133] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 1989.

[134] S. L. Peterson and P. Kearns. Rollback based on vector time. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 68–77, October 1993.

[135] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Department of Computer Science, Princeton University, June 1993.

[136] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proc. Usenix Technical Conference*, pages 213–224, January 1995.

[137] J. S. Plank and K. Li. Faster checkpointing with $n+1$ parity. In *Proceedings of the Twenty Fourth Annual International Symposium on Fault-Tolerant Computing, FTCS-24*, pages 288–297, June 1994.

[138] J.S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.

[139] J. B. Postel. Internet Protocol. Internet Request For Comments RFC 791, September 1981.

[140] M. L. Powell and D. L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proc. 9th ACM Symp. Oper. Syst. Principles*, pages 100–109, October 1983.

[141] A. Prakash and R. Subramanian. Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulation. In *Proc. the 24th Annual Simulation Symposium, 1991 Simulation Multiconference*, pages 123–132, April 1991.

[142] P. Ramanathan and K. G. Shin. Checkpointing and rollback recovery in a distributed system using common time base. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 13–21, 1988.

[143] P. Ramanathan and K. G. Shin. Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Trans. Software Eng.*, 19(6):571–583, June 1993.

[144] B. Randell. System structure for software fault tolerance. *IEEE Trans. Software Eng.*, SE-1(2):220–232, June 1975.

[145] B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance, Ed. M. R. Lyu*, pages 1–21. John Wiley & Sons, 1995.

[146] M. Ruffin. Kitlog: A generic logging service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 139–148, October 1992.

[147] D. L. Russell. State restoration in systems of communicating processes. *IEEE Trans. Software Eng.*, SE-6(2):183–194, March 1980.

[148] M. Russinovich, B. Cogswell, and Z. Segall. Replay for concurrent nondeterministic shared memory applications. *To appear in Proc. SIGPLAN '96*.

[149] M. Russinovich, Z. Segall, and D. P. Siewiorek. Application transparent fault management in fault-tolerant mach. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 10–19, June 1993.

[150] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

[151] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[152] 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, December 1994.

[153] D. D. Sharma and D. K. Pradhan. An efficient coordinated checkpointing scheme for multicomputers. In *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, June 1994.

[154] L. M. Silva and J. G. Silva. Global checkpointing for distributed programs. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 155–162, October 1992.

[155] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, 1992.

[156] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.

[157] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[158] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pages 250–259, June 1996.

[159] J. M. Smith and J. Ioannidis. Implementing remote fork() with checkpoint/restart. *IEEE Technical Committee on Operating Systems Newsletter*, pages 12–16, February 1989.

[160] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 361–370, 1995.

[161] 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*, October 1996.

[162] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 382–388, 1986.

[163] 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*, pages 96–102, April 1989.

[164] G. Stellner. Consistent checkpoints of PVM applications. In *First European PVM User Group Meeting*, 1994.

[165] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium*, April 1996.

[166] R. E. Strom, , S. A. Yemini, and D. F. Bacon. A recoverable object store. In *Proc. Hawaii International Conference on System Sciences*, pages II–215–II–221, January 1988.

[167] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 44–49, 1988.

[168] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, August 1985.

[169] G. Suri, Y. Huang, Y. M. Wang, W. K. Fuchs, and C. Kintala. An implementation and performance measurement of the progressive retry technique. In *Proc. IEEE International Computer Performance and Dependability Symposium*, pages 41–48, April 1995.

[170] G. Suri, B. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 279–288, June 1995.

[171] V.-O. Tam and M. Hsu. Fast recovery in distributed shared virtual memory systems. In *The 10th International Conference On Distributed Computing Systems*, pages 38–45, May 1990.

[172] Y. Tamir and T. M. Frazier. Application-transparent process-level error recovery for multicomputers. In *Hawaii International Conferences on System Sciences-22*, pages 296–305, January 1989.

[173] Y. Tamir and T. M. Frazier. Error-recovery in multicomputers using asynchronous coordinated checkpointing. Technical Report CSD-910066, University of California, Los Angeles, September 1991.

[174] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. Int. Conf. Parallel Processing*, pages 32–41, 1984.

[175] 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*, pages 118–123, 1986.

[176] M. Theimer, K. Lantz, and D. R. Cheriton. Preemptable remote execution facilities in the V-system. In *Proceedings of the 10th SIGOPS Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[177] Z. Tong, R. Y. Kain, and W. T. Tsai. Rollback recovery in distributed systems using loosely synchronized clocks. *IEEE Trans. Parallel and Distributed Syst.*, 3(2):246–251, March 1992.

[178] K. Tsuruoka, A. Kaneko, and Y. Nishihara. Dynamic recovery schemes for distributed processes. In *Proc. IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, pages 124–130, 1981.

[179] N. H. Vaidya. Consistent logical checkpointing. Technical Report # 94-051, Dept. of Computer Science, Texas A&M University, July 1994.

[180] S. Venkatesan. Message-optimal incremental snapshots. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 53–60, 1989.

[181] K. Venkatesh, T. Radhakrishnan, and H. F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25:295–303, July 1987.

[182] Y. M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *To appear in IEEE Trans. on Computers*.

[183] Y. M. Wang. *Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, August 1993.

[184] Y. M. Wang. The maximum and minimum consistent global checkpoints and their applications. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 86–95, September 1995.

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

[186] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. Parallel and Distributed Syst.*, 6(5):546–554, May 1995.

[187] Y. M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 147–154, October 1992.

[188] Y. M. Wang and W. K. Fuchs. Scheduling message processing for reducing rollback propagation. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 204–211, July 1992.

[189] Y. M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 78–85, October 1993.

[190] Y. M. Wang, Y. Huang, and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 138–144, June 1993.

[191] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 22–31, June 1995.

[192] Y. M. Wang, A. Lowry, and W. K. Fuchs. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters*, 50(4):223–230, May 1994.

[193] Y. M. Wang, Michael Merritt, and A. B. Romanovsky. Guaranteed deadlock recovery: Deadlock resolution with rollback propagation. In *Proc. Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 92–97, December 1995.

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

[195] W. G. Wood. A decentralized recovery control protocol. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 159–164, 1981.

[196] W. G. Wood. Recovery control of communicating processes in a distributed system. In *Reliable Computer Systems, Ed. S. K. Shrivastava*, pages 448–473. Berlin, Germany: Springer-Verlag, 1985.

[197] K. L. Wu and W. K. Fuchs. Recoverable distributed shared virtual memory. *IEEE Trans. Comput.*, 39(4):460–469, April 1990.

[198] K. L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Trans. Parallel and Distributed Syst.*, 1(2):231–240, April 1990.

[199] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, pages 754–761, December 1993.

[200] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 499–509, 1995.