Complete Process Recovery in Distributed Systems Using Vector Time

Golden G. Richard III Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
Email: {grichard, singhal}@cis.ohio-state.edu

Abstract: *Distributed applications typically consist of a group of processes executing on processors which share neither a global clock nor shared memory and which communicate solely via message passing. As the number of processors increases, so does the probability that a processor will fail at some point during application execution in a distributed system; this is particularly true for long-running applications. In order to reduce lost work, processes which make up a distributed application should be recoverable.*

This paper presents a comprehensive technique for complete *process recovery in distributed systems. By complete recovery we mean the restoration of a consistent system state as well as proper handling of lost, duplicate, and delayed messages resulting from the failure and restoration of the system to a consistent state. Message handling has often been treated casually in previous recovery techniques. In constrast, our recovery techniques directly address message handling issues. We classify the message types which a recovery mechanism must deal with to motivate our message-handling techniques. The use of vector time allows us to reduce the overhead associated with recovery and helps expedite the recovery process after a failure.*

Keywords: Message passing, distributed process recovery, complete process recovery, vector time, causality, checkpointing, message logging, optimistic recovery.

1 Introduction

A distributed system consists of a set of processors, connected by a network, which communicate by passing messages. Distributed systems provide an attractive alternative to large uniprocessors based on cost, elimination of the central point of failure, and flexibility. Traditionally programs running on a distributed system are structured as communicating processes executing on physically distinct processors. As the number of processors in a distributed system and the running time of applications executing on such a system increase, the likelihood of processor failure due to a hardware malfunction, power failure, user error, etc. during application execution increases. To minimize or eliminate lost work in the event of a processor failure, processes running in a distributed system should be recoverable.

Process checkpointing is used to recover processes from processor failures. When a process takes a checkpoint, the state of the process is saved on stable storage. After a failure, process checkpoints are used to restore the system to a consistent state. A *consistent state* in a distributed system is a state in which no process has recorded a message receive operation in its state for which the corresponding message send operation has not been recorded.

Many previous techniques for process recovery do not directly address all the associated message handling issues, such as restoration of messages lost as a result of process failure, handling of duplicate messages, etc. Complete process recovery addresses not only restoration of the system to a consistent state, but also proper handling of lost, duplicate, and delayed messages. In this paper, we classify the various message types which arise in the context of distributed process recovery. Such a classification provides insight into the various message types that can occur as

a result of failure and recovery. We use this classification to help develop techniques for complete process recovery.

The rest of the paper is organized as follows. In Section 2, we outline the system model and failure semantics assumptions and discuss the techniques which form the basic building blocks of distributed recovery. In Section 3, we classify messages which must be dealt with by recovery algorithms. Section 4 provides a high-level overview of our techniques for complete process recovery and discusses data structures maintained by processes to support recovery. Sections 5 and 6 discuss restoration of the system to a consistent state and the related message management issues, respectively. Optimizations are discussed in Section 7. Related work is discussed in Section 8 and we conclude the paper with final thoughts and directions for future research in Section 9.

2 Background and Motivation

2.1 System Model

The system we consider is composed of a collection of N processors, indexed 1 through N, connected via a network. The processors share neither a physical memory nor a common (global) clock. Processors communicate solely by exchanging messages.

The network is assumed to provide fault-free, FIFO communication with finite message delay between the processors. Processors are assumed to exhibit fail-stop [17] failure semantics; that is, they stop functioning when they fail without generating spurious or incorrect messages. A checkpoint/restore mechanism, capable of saving and restoring the state of processes executing on the processors, is assumed to exist. Time-outs or some alternative mechanism is used to detect processor failures. Individual processes which make up an application are deterministic,

though the application as a whole can be non-deterministic because of variable message propagation times.

2.2 Recovery Techniques

Process checkpointing provides a technique for restoring a distributed system to a globally consistent state after a failure. When a process takes a checkpoint, the state of the process (i.e., the process registers, stack, local memory contents, etc.) is saved on stable storage. After a failure, process checkpoints are used to restore the system to a consistent state. Recall that a *consistent* state in a distributed system is a state in which no individual process state includes a message receive for which the corresponding message send is not included in the state of some other process. A message whose receipt has been recorded in the state of a process but whose send event has been lost is called an *orphan* message. While a message whose receipt has been lost but whose send event has been recorded in the state of a process (called a *lost* message) does not strictly violate this definition of consistency, it must be dealt with nonetheless. Orphan and lost messages will be discussed in greater detail later.

Two methods for taking checkpoints are commonly used: synchronous checkpointing and asynchronous checkpointing. In *synchronous* schemes, processes coordinate among themselves to ensure that their checkpointed states form a consistent global state. Thus, a globally consistent set of checkpoints exists in the system at all times and recovery after a process failure consists of restoring each process' state using its latest checkpoint. To minimize the amount of lost work due to a failure, the global checkpoint is advanced periodically. Processes collect globally consistent checkpoints either by explicit communication during checkpointing (e.g., [9, 10, 11]) or by scheduling checkpointing events based on synchronized processor clocks which simulate a

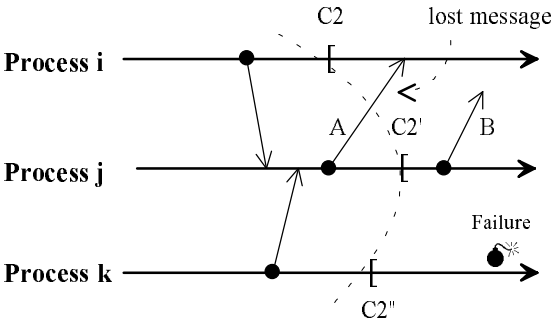

Figure 1. Synchronous checkpointing results in message loss.

global clock (e.g., [3, 15, 24]). When the system is restored to a consistent state, the global state may contain lost messages. For example, in Figure 1, when the processor executing process *k* fails, the system is restored to a consistent state corresponding to (C1, C1', C1"). Message A has been lost because its receive event has been undone by the rollback, but the send event has not. Messages like B can cause system inconsistency when they are sent during normal execution but received after a subsequent failure and checkpoint restoration. Synchronous checkpointing alone does not guarantee that lost messages will be avoided nor does it necessarily provide a mechanism for dealing with delayed messages.

Asynchronous checkpointing (e.g., [1, 28]) requires no interaction between processes during checkpointing. Processes periodically (and independently) take process checkpoints on stable storage. After a failure, processes communicate to choose a globally consistent set of local checkpoints. This set is used to restore the system to a consistent state. Rollback propagation (called the *domino effect*) occurs if no globally consistent set of checkpoints (other than the initial states of all processes) exists. In the extreme case, the propagation results in all processes being rolled back to the beginning of execution. Consider processes *i* and *j* in Figure 2. Process *i* has taken process checkpoints C1, C2, and C3 and process *j* has taken checkpoints C1' and C2'.

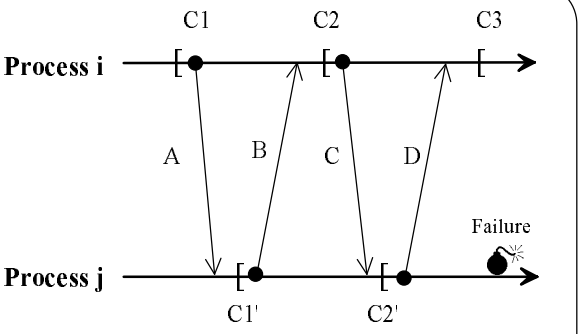

Figure 2. Asynchronous checkpointing and the domino effect.

Process *i* has sent messages A and C and process *j* has sent messages B and D. When process *j*'s processor fails, *j* is restored from its most recent checkpoint, C2'. However, the resulting global state, (C3, C2'), is not consistent because the send of message D has has been undone, though the receipt of D is recorded in the state of process *i*. Process *i* is then rolled back to C2 in an attempt to achieve a consistent system state. The resulting global state (C2, C2') is still not consistent, however, because in the process of rolling back to C2, the send of message C has been undone, though *j* has received this message. Process *j* is then rolled back. This rollback propagation continues until *i* and *j* have been rolled back to the beginning, thus losing all work they performed before the failure.

As with synchronous checkpointing, restoration of the system to a consistent state using asynchronous checkpointing does not guarantee that no messages have been lost nor that duplicate and delayed messages are handled correctly. We will concentrate on asynchronous checkpointing in the remainder of the paper.

Message Logging

Message logging is used to supplement asynchronous checkpointing and help prevent the domino effect. Message logging at the receiver essentially provides a finer-grained rollback

capability than checkpointing alone. It can also be used to reduce the cost of restoring lost messages.

Logging schemes fall into two broad categories, namely, pessimistic and optimistic logging. With *pessimistic* logging, messages are written to stable storage as they are received and before they are processed. This ensures that messages which have been processed are available on stable storage should a failure occur. Domino effect-free recovery for a single process using this strategy is fairly straightforward: the latest checkpoint of the failed process is restored and then messages are replayed from the log on stable storage until the log is exhausted. No other process need roll back. The drawback of pessimistic logging is the potential negative effect on performance during normal system execution, since messages must be forced to stable storage as they are received.

Optimistic message logging (e.g., [20, 22]) takes a more aggressive approach to the problem. As messages are received, they are logged to volatile storage and are periodically flushed to a stable log. This allows the system to use idle time for volatile to stable log flushes and also allows messages to be written in batches, resulting in more efficient disk utilization. However, messages present in the volatile log at the time of a process failure are lost; when the process recovers by restoring its latest checkpoint and replaying messages from the stable log, it may be impossible to avoid orphan messages. As a result, other processes may have to be rolled back to eliminate the orphans. Recovery techniques which use a combination of optimistic message logging and asynchronous checkpointing can significantly reduce overhead during normal system execution. However, they require process interaction during recovery to ensure that the correct

set of checkpoints is chosen and that non-failed process re-execution using the message logs is constrained sufficiently that a globally consistent system state is restored.

2.3 Motivation

As illustrated in Section 2.2, neither synchronous nor asynchronous checkpointing alone address all the issues in complete process recovery; they only restore the system to a consistent state and do not directly handle lost, duplicate, or delayed messages. Introducing message logging to minimize the domino effect does not guarantee proper message handling unless additional mechanisms are used. Many existing recovery techniques address only consistent state restoration or address message-handling issues casually. For example, Koo and Toueg [10] delegate responsibility for handling lost messages to the communication system. In some cases, this assumes far too much about the capabilities of the underlying communication system.

To simplify message handling, we classify the message types which arise during failure and recovery. We develop techniques for complete process recovery which address both consistent state restoration and proper message handling. Our techniques make use of the message logs to reduce the number of lost messages which must be retransmitted.

3 Message Classification in Distributed Process Recovery

Process failure and the restoration of the system to a consistent state may undo the send or receive events (or both) of many messages. This may result in various abnormal situations with respect to those messages which must be dealt with correctly for the system to be recovered completely. For example, detecting messages whose send events are not undone but whose receive events have been undone or lost (called *lost* messages) is important. Handling messages which are in transit during failure but whose send events are undone by the failure is also

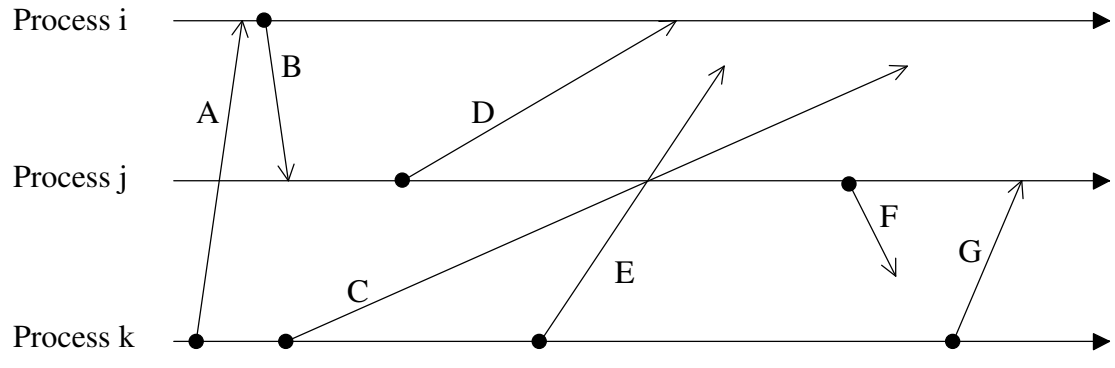

Figure 3. System state at the point of process i's processor failure.

important. To aid in the management of the various message types when designing complete recovery schemes, it is important to classify the types of messages that are induced by process failure and recovery. Classifying message types will help clarify the message management issues which must be addressed when designing such recovery schemes.

We first consider the case where processes are restored after a failure using only checkpoints. In Figure 3, processes *i, j,* and *k* have sent messages A-G and messages A, B, D, and G have been received at the points indicated. Messages C, E, and F are still in transit when the processor executing *i* fails.

An initial attempt at recovery might restore the process states using the latest checkpoints that are available, as shown in Figure 4. Process checkpoints are indicated by "[". Messages A, B, and G will be handled in a correct fashion by restoration of the checkpoints, since the send and receive events for A and B are recorded and both events for G have been completely undone. We call messages A and B *normal* messages and message G a *vanished* message. Messages C, D, E, and F are potentially problematic, however. Since message C is in transit at the time of failure, several possibilities exist: it may arrive when *i* is still down, in which case the message

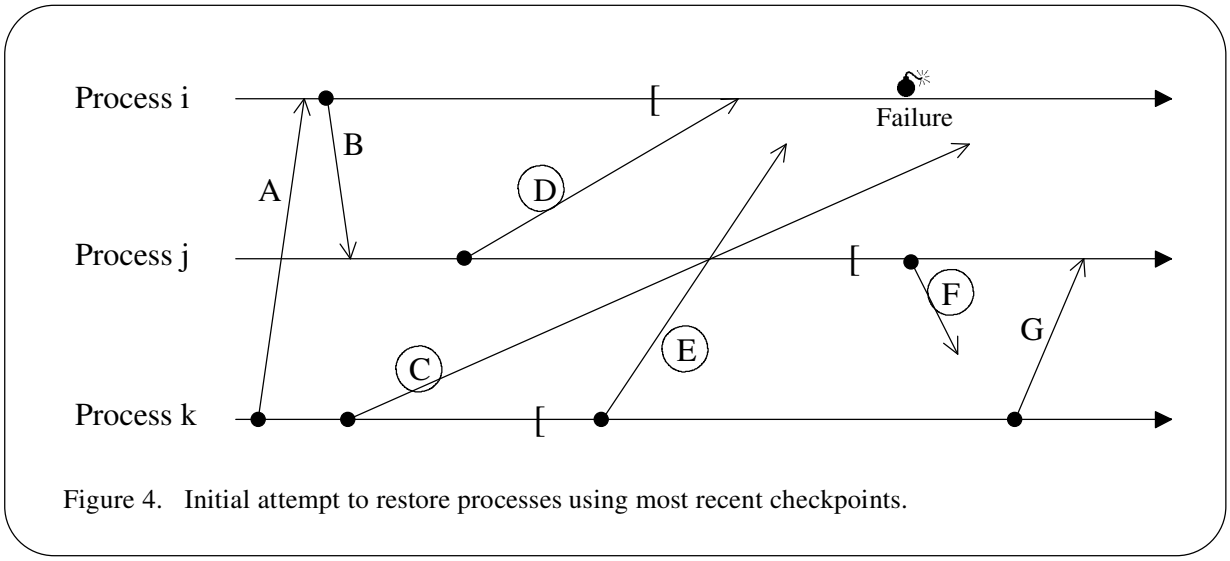

Figure 4. Initial attempt to restore processes using most recent checkpoints.

is discarded; it may arrive when *i* is recovering, or it may arrive after *i* has completed recovery. Each of these scenarios must be dealt with correctly. Message C is an example of a *delayed* message. Message D is a *lost* message, since the send event for D is recorded in the restored state for *j* but the receive event has been undone at *i*. Note that *j* will *not* resend D without an additional mechanism, since the send of D at *j* occurred before the checkpoint and the communication system successfully delivered D before the processor executing *i* failed. Messages E and F pose perhaps the most serious problem of all; when they arrive at their respective destinations either during recovery or after recovery is complete, they must be discarded since their send events have been undone. They are both *delayed* and *orphan* messages. Both of these messages will be regenerated as necessary after processes resume execution from their checkpoints. Recovery techniques must be able to distinguish between messages like C and messages like E and F.

Complete recovery will involve:

• Ensuring that the reception of message C is recorded in the state of process *i* after recovery is complete. This may involve retransmission of the message by process *k*. Whatever mechanism

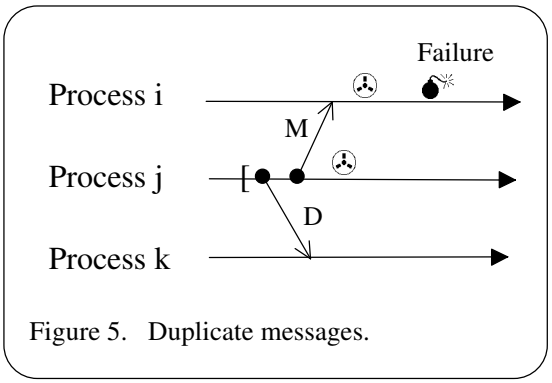

Figure 5. Duplicate messages.

is used, it must distinguish between delayed messages which are not orphans (e.g., message C) and delayed messages which are orphans (e.g., E and F).

• Restoring message D (the communication mechanism will not perform this function). This is likely to require retransmission of the message by process *j*.

• Ensuring that messages E and F are discarded when they are received by *i* and *k,* respectively.

Note that consistent process checkpointing alone handles only the removal of *orphan* messages (discussed earlier). Additional techniques must typically be introduced to handle the situations discussed above.

Generally, asynchronous checkpointing is supplemented with message logging to provide a finer-grained rollback capability. The introduction of message logging and message replay during recovery can introduce *duplicate* messages. In Figure 5, process *j* has sent message D to process *k* and message M to process *i*. After process *i* fails, it recovers to the point indicated by the tape symbol. Assume that process *j* can recover to the point indicated by restoring its checkpoint and replaying its message log to achieve the system state marked by the tape symbol (messages *j* has received are not shown for clarity). In the process of recovery, messages M and D are regenerated. Process *k* has now received the *duplicate* message D twice. To ensure correct

system operation, duplicate messages must be properly handled. We discuss specific techniques for managing the various message types described above in Section 6.

4 A Technique for Complete Process Recovery in Distributed Systems

We now present a technique for complete process recovery in distributed message-passing systems which has low message complexity, does not introduce significant overhead during normal execution, and allows concurrent process rollback after a failure. The proposed technique is based on asynchronous checkpointing with optimistic message logging at the receiver and explicitly addresses lost messages, delayed messages, duplicate messages, etc. rather than concentrating solely on orphan message removal. Our technique requires O(N) messages for restoration of a consistent system state plus the retransmission of some messages lost as a result of failure and recovery. Intelligent log processing allows messages lost by processes other than the failed process to be recovered without retransmission. For clarity, we assume that each processor executes a single process. Processor failure and process failure are, therefore, synonymous in what follows. Making each processor a *recovery unit* [22] allows this restriction to be removed easily. The proposed technique tolerates failure of single processors; we assume that no further failure occurs during recovery.

4.1 Basic Idea

Complete process recovery requires that two issues be addressed: restoration of the system to a consistent state (i.e., elimination of orphan messages) and proper handling of lost, duplicate, and delayed messages. A number of existing recovery techniques do not directly address the latter. Our technique not only restores the system to a consistent state, but also properly handles messages that are left in abnormal states due to the rollback of processes.

Our technique restores the system after a failure as follows. When a process fails and restarts, it restores its latest checkpoint, replays the contents of its stable log, and then broadcasts information about the extent of its recovery to the other processes in the system. The other processes then rollback to an appropriate checkpoint (if orphan messages are detected) and read messages from their logs to advance their states to a point which is consistent with that of the failed process. Messages sent to the failed process which were subsequently lost because of the failure are properly restored. Log processing by the non-failed processes proceeds in such a way that lost messages from non-failed processes to other non-failed processes do not require retransmission to be restored. Delayed and duplicate messages are properly handled so that system inconsistency does not result after recovery is complete.

4.2 Data Structures

Each process *i* maintains a volatile and stable message log as well as a vector timestamp *TSi* and variable *INCi* which contains the process' view of the system's global incarnation number. A vector *DVi* is used to track direct process dependencies so duplicate messages can be eliminated (*DVi* is discussed in detail in Section 6). The vector timestamp tracks process *i*'s causal dependence on other processes. The incarnation number tracks the number of process failures the system as a whole has experienced; the current value for process *i* is always one more than the total number of process failures of which process *i* is aware. Each process *i* also maintains a set *TINi* containing <*TS, INC*> pairs, initially equal to ∅. *TINi* allows the process *i* to properly handle delayed messages during recovery (this is discussed in further detail in Section 6). *INCi* and *TINi* must be maintained on stable storage so they survive process failures. The current values of *TSi* and *DVi* are lost when a process fails, but the values are stored in each process checkpoint

and therefore values which were current at the time the checkpoint was taken are restored when the checkpoint is used.

When a process sends a message, the pair <*TSi*, *INCi*> is appended to the message (notationally, *m.TS* and *m.INC* will be used to refer to the message *m*'s timestamp and associated incarnation number, respectively). The vector timestamp *TSi* is updated before being attached to a message by incrementing the *i*th element of *TSi* and when *i* receives a message *m* as *TSi* = ***sup****(TSi, m.TS)*[1]. The incarnation number *INCi* is incremented whenever *i* fails and then begins recovery.

4.3 Failure-free Operation

Under normal circumstances, each message sent by a process is tagged with the vector timestamp and incarnation number of the sending process as indicated in the previous section. As messages are received by a process, they are logged to volatile storage and periodically spooled to stable storage in the background. When a process fails, messages in the volatile log are lost, while messages in the stable log are not. Periodically, processes independently take checkpoints.

As indicated in Section 4.1, complete process recovery consists of two important steps: restoration of the system to a consistent state, and message management (handling of duplicate messages, delayed messages, and retransmission or regeneration of messages which are discovered to be "lost" as a result of the failure and recovery). We present a technique for restoring the system to a consistent state in the next section. In Section 6, we address the associated message handling issues which ensure that messages are handled correctly.

---

[1] *sup(TS1, TS2)* returns a vector *TS* such that $\forall k$, *TS[k] = max(TS1[k], TS2[k])*.

5 Restoring the System to a Consistent State

A process *i* performs the following steps to recover after a crash. Messages received during recovery are queued until recovery is complete.

1. Process *i* restores its latest checkpoint (which includes the vector timestamp *TSi* and direct dependency vector *DVi*) from stable storage and increments the incarnation number *INCi*.

2. Process *i* replays the message log on its stable storage until the log is exhausted, updating *TSi* as messages are processed. When the log contents have been exhausted, the pair <*TSi, INCi*> is broadcast to all other processes in the system in a message called a *Recovery Initiation Message* (RIM).

3. Process *i* now awaits an acknowledgement from every other process. The messages received and queued during recovery are not replayed until the acknowledgements are received. When the acknowledgements are received, process *i* has completed recovery and resumes normal execution.

When a process *j* receives a RIM containing <*TSi, INCi*>[1] from the failed process *i*, it performs the following steps. Messages received during recovery are queued until recovery is complete.

1. *INCj* is set to *INCi*.

2. <*TSi, INCi*> is added to the set *TINj*.

[1] *TSi* and *INCi* are the current vector timestamp and incarnation number for process *i*, respectively.

3 If $TSi < TSj$ (i.e., *j* causally depends upon a state which has been rolled back), then *j* must roll back and proceeds as follows:

i Process *j* rolls back by restoring its latest checkpoint *CKj* such that $CKj.TS < TSi$ (i.e., the latest checkpoint for process *j* whose associated timestamp is not greater than the timestamp associated with the failed process' state).

ii Process *j* replays messages from its log, updating *TSj*, until a message *m* is encountered with $m.TS > TSi$. The remaining log contents are not replayed and must be deleted from volatile and stable storage.

4 If $TSi \geq TSj$, then *j* does not need to roll back.

5 Process *j* sends an acknowledgement to the failed process *i* to indicate it has completed recovery and then resumes normal execution.

5.1 Correctness Proof

We now show that after every process in the system has executed the recovery procedure, the system is restored to a consistent global state.

Lemma 1. *When the rollback/replay phase is complete at each process, the state of the failed process is pairwise-consistent with the states of each of the non-failed processes in the system.*

Proof. We must show there are no orphan events recorded (where the failed process is one of the processes involved in the message transmission) when the rollback/replay phase is complete. Assume the contrary. Let process *i* be the failed process, process *j* a non-failed process. There are two possible cases.

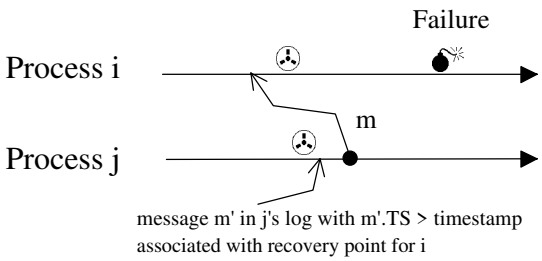

Figure 6. Orphan impossibility: Scenario 1.

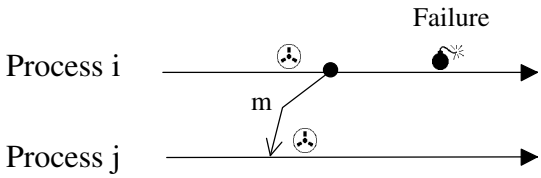

Figure 7. Orphan impossibility: Scenario 2.

Case 1: There exists a message *m* whose receipt is recorded in the state of the failed process *i*, but whose send event is not recorded in the state of the non-failed process *j* which sent it. Figure 6 depicts this scenario. Since process *j* did not re-execute the send of message *m* during rollback/replay, there must be a message *m'* in its message log whose reception precedes the send of *m* but and whose *m'.TS* is greater than the timestamp associated with the recovery point of *i*. This is clearly impossible, since then message *m* would have been assigned a timestamp greater than *m'* and process *i* would have distributed a recovery point timestamp to *j* at least as large as *m.TS*, allowing the message *m* to be re-sent, eliminating the orphan condition.

Case 2: There exists a message *m* whose receipt is recorded in the state of the non-failed process *j*, but whose send event is not recorded in the state of the failed process *i* which sent it. Figure 7 shows this scenario, in which process *j* has processed a message *m* with *m.TS* greater than the timestamp associated with the point of recovery for *i*. This clearly violates the recovery protocol outlined in the previous section because *j* should not have processed this message. □

Lemma 2. *When the rollback/replay phase is complete at each process, the states of the non-failed processes in the system are pairwise consistent.*

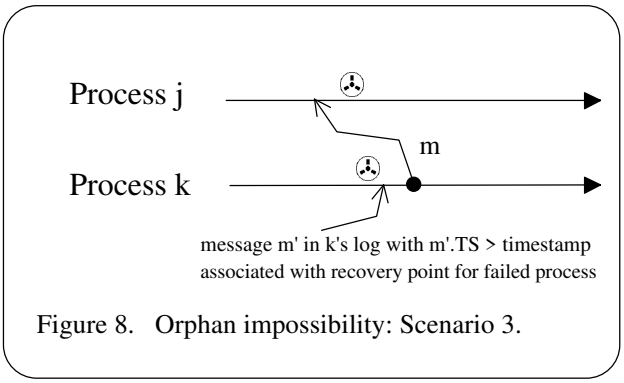

Figure 8. Orphan impossibility: Scenario 3.

Proof. We must show there are no orphan events recorded in the process states (where both the send and receive events occurred in non-failed processes) when the rollback/replay phase is complete. This case is similar to that considered in Case 1 of Lemma 1. Assume the contrary. Let *j* and *k* be two arbitrary non-failed processes such that there exists a message *m* whose receipt is recorded in the state of the process *j*, but whose send event is not recorded in the state of the process *k*, which sent it. Figure 8 illustrates this situation. Since process *k* did not re-execute the send of message *m* during rollback/replay, there must be a message *m'* in its message log whose reception precedes the send of *m* and with *m'.TS* is greater than the timestamp associated with the recovery point of the failed process (not shown in the figure). However, if this is true, then *m.TS* is greater than *m'.TS*, since the send of *m* is causally dependent on the receipt of *m'*. Then *j* could not have processed message *m* in its log during rollback/replay, since *m* must have a timestamp greater than the timestamp associated with the recovery point of the failed process. □

Theorem 1. *The global system state restored by the rollback/replay phase is consistent.*

Proof. The proof follows directly from Lemmas 1 and 2. □

6 Message Management

Processor failure and the subsequent restoration of the system to a globally consistent state can result in duplicate, vanished, and lost messages being introduced. Delayed messages can also complicate the recovery procedure. In this section we discuss techniques which ensure that delayed, duplicate, vanished, and lost messages are handled correctly. These techniques, together with the consistent state restoration technique discussed in Section 5, form a coherent method for complete process recovery.

Delayed messages

Messages whose send and receive events occur in different system incarnations (called *delayed* messages) can be problematic. Messages C, E, and F in Figure 4 are examples of delayed messages. When a message *m* is received by a process *j, m.TS* and *m.INC* are examined to determine if *m* should be processed or discarded using the following rule:

Rule 1:

Message m is discarded by process j iff $\exists \langle TS,INC \rangle \in TINj \ .\ni. \ m.TS > TS \wedge m.INC < INC$

If the condition above holds, it means that *m* was sent by a process which subsequently had to roll back to a state prior to the send of *m*. Under such circumstances, *m* must be discarded because the send event for *m* has been undone.

Duplicate messages

Duplicate messages (depicted in Figure 5) arise when a message is sent multiple times because of rollback and message log replay. Duplicates can be suppressed at the receiving end by having each process track the highest state number seen directly from every other process in the system. To accomplish this, each process *i* maintains a vector *DVi* which tracks *i*'s direct dependency on

other processes [20]. *DVi* is initially set to all zeros. Each time process *i* receives a message *m* from a process *j*, *DVi[j]* is set to the maximum of *DVi[j]* and *m.TS[j]*. To guarantee that duplicate messages are discarded, a process uses the following rule:

Rule 2:

A message m received by process i from a process j is discarded if m.TS[j] < DVi[j]. Otherwise m is processed normally.

Since the vector *DVi* captures *i'*s direct dependencies on the process states of the other processes, a message received from a process with a state number lower than *i* has previously seen from that process is certainly a duplicate. Note that FIFO communication channels are necessary for this rule to be applicable.

Note that messages generated by the failed process during replay of its stable message log (up to the point of the last processed message in the log prior to failure) are either in transit or contained in the logs of the non-failed receiver processes. The non-failed processes are able to process these messages without their retransmission by the failed process. This will reduce the number of duplicate messages transmitted. Messages generated by non-failed processes during their log replay cannot simply be suppressed in this manner since the failed process may not have logged all messages before failing.

Lost messages

We now address *lost* messages (messages whose receive events have been rolled back). Message D in Figure 4 is an example of this type of message. The following lost message categories must be addressed:

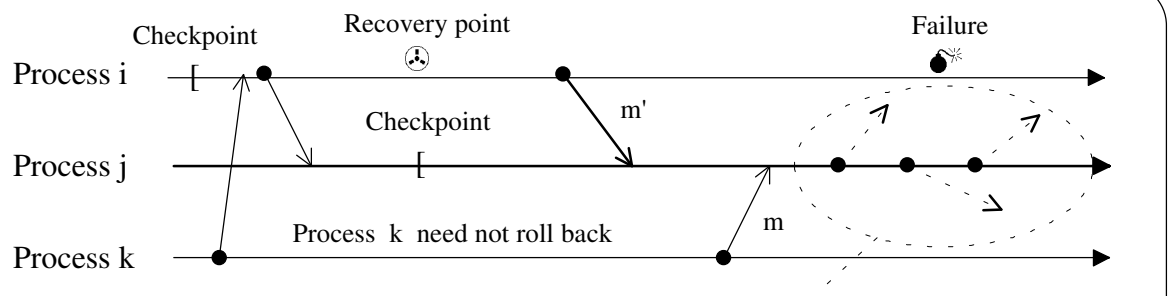

Figure 9. Category 1 lost message.

1 Messages sent by a non-failed process to other non-failed processes whose send events precede the checkpoint used to restore the process (if the non-failed sender was not required to rollback, we regard its current state as the "restored" state).

2 Messages sent by a non-failed process to the failed process whose send events precede the checkpoint used to restore the non-failed process.

Messages sent by a process whose send events follow the checkpoint used to restore the process need no special treatment; they will be regenerated as needed during log replay and afterward during normal execution.

More sophisticated log processing than that presented in Section 5 is required to allow restoration of lost messages in category 1 without retransmission. Consider Figure 9. Process *i* has failed at the point indicated and restored its state to the recovery point shown. Process *k* need not roll back at all since it is not causally dependent on the state of *i*. Process *j*, however, must roll back in response to *i*'s failure. Process *j* restores the checkpoint shown and begins log replay to recover. Log processing by *j* ceases and the remaining log contents are deleted when it encounters the orphan message *m'*. However, this results in the loss of message *m* since *k* was not

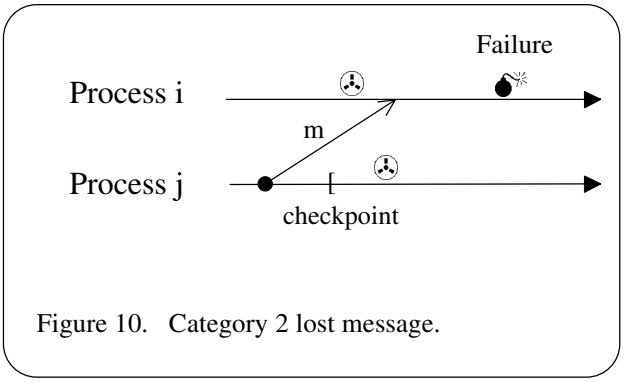

Figure 10. Category 2 lost message.

required to roll back in response to *i*'s failure and thus will not resend *m* automatically. It would be inefficient to force *k* to resend *m* since it is already present in *j*'s log, so we propose the following solution: Process *j* deletes the orphan *m'* when it is encountered and processes *m* normally. The following rule provides a general method for dealing with category 1 lost messages:

Rule 3:

Orphan messages encountered by a non-failed process in its message log during log replay can be deleted to allow processing of messages which causally precede or are concurrent with the recovery point of the failed process.

By deleting messages which causally depend on the recovery point of the failed process (i.e., orphan messages) during log replay, we effectively reorder the log during recovery. This does not introduce any problem because message arrivals from different processes have non-deterministic characteristics. The reordering of messages in the log produces a *potentially different* but valid application execution after recovery. Any messages which depend on the original reception order of *m* and *m'* (for example, the messages in the dotted area in Figure 9) causally depend on the reception of *m'*; these messages will become *vanished* messages during recovery. This is true because *m'* has a timestamp greater than the recovery point of the failed process. Any message sent by *j* after the reception of *m'* has an even larger timestamp and thus will have

both its send and receive events rolled back. Our technique takes direct advantage of the fact that processes already perform receiver-based logging to handle the domino effect. By carefully processing messages in the logs of non-failed processes, we are able to avoid detection or retransmission of category 1 lost messages.

It remains to address lost messages in category 2. Consider Figure 10. If process *j* is restored from the checkpoint indicated in response to *i*'s failure, message *m* is a lost message since its reception is not recorded in the state of the failed process *i*, but the corresponding send event is recorded in the state of process *j*. Note that execution forward from the checkpoint by *j* will not cause retransmission of *m*. If process *j* examines the timestamp of the recovery initiation message from *i* which initiated *j*'s rollback [14], *j* can detect that message *m* needs to be resent. The following rule discusses how to identify such messages and when they should be retransmitted:

Rule 4:

When a process j receives a RIM containing <INCi, TSi> from a failed process i, j must resend any messages m previously sent to i which satisfy TSi[j] < m.TS[j] < TSj[j]. The messages should be sent by j after restoration of an appropriate checkpoint but before message log replay.

The rule indicates that messages sent prior to the checkpoint used to a restore a non-failed process whose receptions are not recorded by the failed process need to be retransmitted. The messages to be resent can be regenerated by rolling the non-failed process back to an earlier checkpoint and reexecuting until the needed messages are generated, or, for better performance, sender-based message logging (to a volatile log only) can be used. Due to space considerations, we do not discuss this issue further here.

Notice our techniques *only* retransmit messages which have been lost by the failed process and messages which are generated by normal execution after recovery.

Vanished messages

Recall that *vanished* messages are messages whose send *and* receive events have been undone or lost by failure/recovery. Message G in Figure 4 is an example of this type of message. Since the send events for vanished messages always follow the recovery points for the processes involved in recovery, they will be regenerated as needed by normal system execution once recovery is complete. No special handling is required.

7 Optimizations

7.1 Checkpoint and Log Garbage Collection

Since stable logs can grow without bound and checkpoints can accumulate, it is necessary to discard obsolete log information and checkpoints to reduce storage requirements. When it can be established that the state of a process need never be rolled back before a checkpoint *CK,* all checkpoints *CK′* for the process with $CK'.TS \prec CK.TS$ can be discarded. Similarly, all logged messages *m* such that $m.TS \prec CK.TS$ can be discarded. To accomplish this, processes can periodically exchange information about the extent of their stable logging. A process *i* can be certain that it will never roll back prior to a state with timestamp *TS* if all messages whose timestamps causally precede *TS* have been logged to stable storage.

7.2 *TINi* Garbage Collection

To reduce both memory usage and the processing time associated with message reception, the *TINi* structure should not be allowed to grow without bound. Obsolete entries in *TINi* can be

discarded in the following manner: An element $<TS, INC> \in TINi$ can be safely deleted from *TINi* when a message *m* has been received by *i* from every process in the system with $m.INC > INC$. Since channels are FIFO, messages from a process with lower incarnation numbers are always received before messages with higher incarnation numbers.

7.3 Reducing Overhead Associated with Vector Timestamps

Managing vector time by attaching the entire vector timestamp for a process to every outgoing message can be expensive if the number of processes in the system is large. Singhal and Kshemkalyani present techniques in [19] which can significantly reduce the overhead associated with vector time maintenance by transferring only portions of the vector timestamp which have changed since the last communication. The techniques are particularly effective when communication is localized.

8 Related Work

Only Peterson and Kearns [14] have previously used vector time to coordinate process rollback after a failure. Our technique uses broadcast rather than token-passing on a ring of processors to allow processes to perform concurrent rollback/recovery in response to a failure. This allows much faster recovery, especially if the number of processors involved is large. In addition, our message-handling techniques are more comprehensive (in particular, lost messages of the type depicted in Figure 9 are problematic for the technique described in [14]).

Koo and Toueg's [10] recovery technique uses synchronous checkpointing. They assume that lost messages are handled by the communication system, while our technique processes messages correctly without this assumption. The synchronous checkpointing technique of Kim and Park [9] has a more relaxed coordination phase, resulting in less runtime overhead than that in

[10]. Leu and Bhargava [11] introduce a synchronous checkpointing technique which permits simultaneous execution by multiple processes and does not require FIFO channels. Non-orphan message-handling is not addressed directly. All these techniques ([9], [10], and [11]) allow non-deterministic processes since they do not rely on message logging/replay. Silva and Silva [18] present a non-blocking synchronous checkpointing scheme which uses limited logging to eliminate lost messages. Techniques by Venkatesh et al. [26] and Briatico et al. [2] allow processes to take independent checkpoints but make processes take additional checkpoints as necessary (synchronously with message reception) to ensure that no orphan or lost messages exist in the global checkpoint set. Wang and Fuchs [29] introduce a *lazy* checkpointing protocol which potentially allows some inconsistent checkpoint sets (in order to reduce the overhead associated with consistent checkpointing), but avoids the domino effect. Li, Naughton, and Plank [12] introduce a synchronous checkpointing technique for multicomputers, based on a central checkpoint coordinator, which does address message-handling issues. Their technique is not applicable to distributed systems in general since it relies on properties of multicomputer message routing, in particular, a deterministic, deadlock-free routing function. This allows delayed messages to be flushed more quickly than in general purpose, fully-connected networks of distributed processors. Wojcik and Wojcik's [27] recovery technique is synchronous, but avoids blocking during checkpointing by having the sender and receiver balance the messages which have been sent and received. Wang and Fuchs [28] present an unsynchronized checkpointing technique with an aggressive checkpoint reclamation algorithm. Their technique is susceptible to the domino effect since checkpoints are not coordinated, but non-deterministic processes are permitted.

Sistla and Welch's [20] recovery algorithms initiate a recovery phase after processor failure which involves all processes exchanging information about the extent of their respective logged states. Processes then rollback and replay messages to achieve the maximum possible globally consistent state. Duplicate messages are handled directly and delayed messages are handled implicitly because of the coordination at the time of failure. Lost messages are not addressed. Their techniques have $O(N^2)$ and $O(N^3)$ message complexity.

Bhargava and Lian's [1] recovery technique uses a two-phase rollback algorithm. It minimizes overhead during normal execution and is therefore appropriate for environments where failure is infrequent. In the worst case, processes may have to roll back to the beginning of execution. Strom and Yemini's [22] technique is based on optimistic message logging and tolerates failure of an arbitrary number of processors. In the worst case, $O(2^N)$ process rollbacks may occur, though orphan, duplicate, and lost messages are handled[1]. Strom et al. [23] discuss enhancements to this work which potentially reduce logging overhead.

Juang and Venkatesan [8] describe methods for recovering processes which attach very little (or no) additional information to each application message. Establishment of a consistent global state is accomplished via a coordinated effort at recovery time. Venkatesan's [25] recovery techniques require processes to pessimistically log the sender ID of messages that are received. Upon failure, a process recovers by requesting that lost messages be resent. Non-failed processors need not roll back.

Johnson and Zwaenpoel [6] present an optimistic message logging and checkpointing method for message-passing systems based on a central server which monitors the logging progress of

[1] Our terminology differs from that of Strom and Yemini; they define *lost* message in a broad way that includes both *orphans* and the messages we define as *lost*.

the system and advances the global recovery state as processes log messages to stable storage. Handling of non-orphan messages is delegated to the communication system. Johnson [7] introduces a technique which allows processes to commit states to stable storage as needed to ensure fast output to the outside world and which handles both deterministic and non-deterministic processes.

Elnozahy and Zwaenpoel [4] discuss the Manetho rollback recovery protocol, which is based on *antecedence graphs*. When a process fails and recovers, it utilizes the antecedence graph and communicates with other processes in order to regenerate a consistent state. The recovery method is complex, but non-failed processes need not rollback.

Ramanathan and Shin [15], Tong et al. [24], and Cristian and Jahanian [3] discuss recovery using synchronized physical clocks. The first uses additional hardware to aid in clock synchronization while the latter two use software-only approaches.

9 Conclusions and Future Work

We have presented a comprehensive, application-transparent recovery technique for distributed message-passing systems which uses vector time and addresses both consistent state restoration and proper handling of application messages left in unorthodox states as a result of failure and recovery. Classifying the types of messages that occur during process recovery allows us to more easily identify and address the message-handling issues which arise when designing recovery techniques. Our technique extends asynchronous checkpointing and optimistic message logging to allow *complete* process recovery.

Our technique requires processes to roll back at most once in response to a failure and restores the system to the maximal strongly consistent state. After a failure, non-failed processes in the

system rollback and recover *concurrently* as necessary. The technique has substantially lower message complexity (O(N), where N is the number of processes in the system) than many other checkpointing/logging-based recovery techniques. We discuss efficient methods of dealing with *duplicate*, *delayed*, *lost*, and *vanished* messages which integrate with our consistent state restoration technique to achieve complete process recovery. Our log processing method allows the message logs to be used to eliminate the need to retransmit many messages.

We are currently investigating extensions to complete process recovery to handle multiple failures, including failures which occur while recovery is in progress.

Acknowledgments

Thanks to Ann Richard for suggesting the term *vanished message* to refer to messages whose send and receive events have been undone by failure and recovery.

References

[1] B. Bhargava, S. R. Lian, "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems -- An Optimistic Approach," *Proc. of the 7th Symp. on Reliable Distributed Systems*, Columbus, OH, pp. 3-12, October 1988.

[2] D. Briatico, A. Ciuffoletti, L. Simoncini, "A Distributed Domino-Effect Free Recovery Algorithm," *4th Symposium on Reliability in Distributed Software and Database Systems*, Silver Springs, Maryland, pp. 207-215, October 1984.

[3] F. Cristian, F. Jahanian, "A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations," *Proc. of the 10th Symp. on Reliable Distributed Systems,* Pisa, Italy, pp. 12-20, October 1991.

[4] E. N. Elnozahy, W. Zwaenpoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit," *IEEE Trans. on Computers,* vol. 41, no. 5, pp. 526-531, May 1992.

[5] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," *Australian Computer Science Communications*, vol. 10, no. 1, pp. 56-66, February 1988.

[6] D. B. Johnson, W. Zwaenpoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms*, No. 11, pp. 462-491, 1990.

[7] D. B. Johnson, "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 86-95, October 1993.

[8] T. Y. Juang, S. Venkatesan, "Crash Recovery With Little Overhead," *IEEE 11th Intl. Conf. on Distributed Computing Systems,* 1991.

[9] J. L. Kim, T. Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems,* vol. 4, no. 8, pp. 955-960, August 1993.

[10] R. Koo, S. Toueg, "Checkpointing and Rollback Recovery for Distributed Systems," *IEEE Trans. on Software Engineering,* vol. SE-13, no. 1, pp. 23-31, January 1987.

[11] P-J. Leu, B. Bhargava, "Concurrent Robust Checkpointing and Recovery in Distributed Systems," *Proc. of the 4th Intl. Symp. on Data Engineering,* Los Angeles, CA, pp. 154-163, February 1988.

[12] K. Li, J. F. Naughton, J. S. Plank, "Checkpointing Multicomputer Applications," *Proc. of the 10th Symp. on Reliable Distributed Systems,* Pisa, Italy, pp. 2-11, October 1991.

[13] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, pp. 215-226, North-Holland, 1989,

[14] S. L. Peterson, P. Kearns, "Rollback Based on Vector Time," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 68-77, October 1993.

[15] P. Ramanathan, K. G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System," *IEEE Trans. on Software Engineering,* vol. 19, no. 6, pp. 571-583, June 1993.

[16] G. Richard, M. Singhal, "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory," *Proc. of the 12th Symp. on Reliable Distributed Systems,* Princeton, NJ, pp. 58-67, October 1993.

[17] R. D. Schlichting, F. B. Schneider, "Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems," *ACM Trans. on Computer Systems*, vol. 1, no. 3, pp. 222-238, August 1982.

[18] L. Silva, J. Silva, "Global Checkpointing for Distributed Programs," *Proc. 11th Symp. on Reliable Distributed Systems*, pp. 155-162, October 1992.

[19] M. Singhal, A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, pp. 47-52, August 1992.

[20] A. P. Sistla, J. L. Welch, "Efficient Distributed Recovery Using Message Logging," *Proc. of the 8th Annual ACM Symp. on Principles of Distributed Computing*, August 1989.

[21] S. H. Son, A. K. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Trans. on Software Engineering*, vol. 15, no. 10, October 1989.

[22] R. E. Strom, S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. on Computer Systems*, vol. 3, no. 3, pp. 204-226, August 1985.

[23] R. E. Strom, D. Bacon, S. Yemini, "Volatile Logging in n-Fault-Tolerant Distributed Systems," *Proc. IEEE Int'l. Symposium on Fault Tolerant Computing,* pp. 44-49, 1988.

[24] Z. Tong, R. Kain, W. Tsai, "Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks," *IEEE Trans. on Parallel and Distributed Systems,* vol. 3, no. 2, pp. 246-251, March 1992.

[25] S. Venkatesan, "Optimistic Crash Recovery Without Rolling Back Non-Faulty Processors," Tech Report IS-92-813-A, University of Texas at Dallas, 1992.

[26] K. Venkatesh, T. Radhakrishnan, H. F. Li, "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery," *Information Processing Letters,* pp. 295-303, July 1987.

[27] Z. Wojcik, B. Wojcik, "Fault Tolerant Distributed Computing Using Atomic Send-Receive Checkpoints," *Proc. of the 2nd IEEE Symposium on Parallel and Distributed Processing,* pp. 215-222, 1990.

[28] Y-M. Wang, W. K. Fuchs, "Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems," *Proc. of the 11th Symp. on Reliable Distributed Systems,* pp. 147-154, October 1992.

[29] Y-M. Wang, W. K. Fuchs, "Lazy Checkpoint Coordination for Bounding Rollback Propagation," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 78-85, October 1993.