

**Computer Science Department Technical Report  
University of California  
Los Angeles, CA 90024-1596**

**ERROR-RECOVERY IN MULTICOMPUTERS USING  
ASYNCHRONOUS COORDINATED CHECKPOINTING**

**Y. Tamir  
T. Frazier**

**September 1991  
CSD-910066**



# Error-Recovery in Multicomputers Using Asynchronous Coordinated Checkpointing †

*Yuval Tamir and Tiffany M. Frazier*

Computer Science Department  
4731 Boelter Hall  
University of California  
Los Angeles, California 90024-1596  
U.S.A.  
Phone: (213)825-4033 E-mail: tamir@cs.ucla.edu

## Abstract

One of the main approaches to application-transparent error-recovery in multicomputers is based on coordinated checkpointing of interacting sets of processes. This approach does not require logging of messages or transmission of special bookkeeping information with each message. Hence, the performance overhead for fault tolerance during normal operation is minimized. The main disadvantage of previously published schemes based on this approach is the requirement of *synchronous* checkpointing — while processes are being checkpointed to stable storage, their normal execution has to be suspended. We propose a new distributed checkpointing and recovery scheme, based on *asynchronous* coordinated checkpointing of interacting sets of processes, which allows processes to proceed with normal computation *during* checkpointing. Disruption to normal computation and the performance penalty are thus minimized. At each node, checkpointing involves copying the changed process state to local volatile storage followed by resumption of normal computation while the interacting set is identified and checkpointed into stable storage. An enhanced virtual memory system is used to determine which part of each process state has been modified since the last checkpoint and minimize physical data movement during checkpointing.

**Index Terms:** Checkpointing, distributed error recovery, distributed systems, fault tolerance, transparent recovery.

---

† This research is supported by Hughes Aircraft Company and the State of California MICRO program.

## I. Introduction

*Multicomputer* systems, consisting of hundreds or thousands of processors interconnected by point-to-point links, are now technologically and economically feasible [7, 18, 26]. Such systems can achieve high performance at a low cost by exploiting parallelism. Even for “general-purpose” applications, the reliability requirements of large multicomputers, implemented with thousands of VLSI chips, can only be met by using fault tolerance techniques [23]. Thus, the system must be able to detect errors, recover a valid system state, and continue normal correct operation [15]. Since the goal of multicomputers is to achieve high performance, the fault tolerance scheme must not slow down the system significantly, even with applications that are “communication-intensive” (exploit fine-grain parallelism). Given the complexity of application software for large multicomputer systems, it is highly desirable to use a fault tolerance scheme that does not add to this complexity, i.e., a scheme that is *application-transparent*.

We present a new application-transparent low-overhead fault tolerance scheme which allows the system to recover from multiple simultaneous process failures. The scheme involves periodically saving the state of processes in stable storage [13] (*checkpointing*) and *rollback* to the last saved state when recovery is necessary. While the use of checkpointing and rollback for error recovery in distributed systems is not new [15, 14], most of the earlier general-purpose schemes were susceptible to the *domino effect* which, in the worst case, can cause all processes in the systems to roll back to their initial state [15]. The scheme presented in this paper follows the *coordinated checkpointing* approach [1, 9, 12, 23, 24] to application-transparent recovery which is immune from the domino effect. Multiple processes are checkpointed “*simultaneously*” (during a single *checkpointing session*) so that their saved states are consistent [5] and the domino effect cannot occur during recovery.

In previously proposed recovery schemes based on coordinated checkpointing the processes being checkpointed are blocked during the entire checkpointing session [12, 23, 24]. This periodic *synchronous* checkpointing disrupts normal system operation and results in significant performance overhead. The key innovative feature of the *asynchronous* recovery scheme proposed in this paper is the use of *volatile checkpoints* to minimize disruption to normal operation due to checkpointing. A volatile checkpoint is simply a copy of the process state in the node’s local volatile memory. Checkpointing begins with copying the changed state of a process to local volatile storage, after which the process may resume normal computation. As discussed in Section X, volatile checkpoints allow much of this local copying to be avoided through the use of a slightly modified virtual memory system. The rest of a checkpointing session involves identification of the set of processes that are to be checkpointed “simultaneously” and transfer of the volatile checkpoints to *stable storage* (e.g., *disk nodes* [23]). Thus, the process is

suspended during only a small fraction of the time it takes to complete a checkpointing session.

The proposed scheme uses checkpointing and recovery algorithms that involve only as much of the system as is necessary: a set of processes that have interacted, directly or indirectly, since their last checkpoint [1, 12, 24]. There is no need for system-wide central coordination. Processes which are not part of this *interacting set* do not participate in checkpointing/recovery and may continue to do useful work. Unrelated checkpointing and recovery sessions do not interfere with each other, while the actions of related sessions are properly coordinated. Checkpointing results in the saving of a consistent *snapshot* of the states of an interacting set of processes in such a way that a valid global checkpoint of the system state is maintained in stable storage [5, 23, 24].

This paper presents a complete fault tolerance scheme. The proposed error recovery technique is used in conjunction with practical, low overhead, error detection mechanisms. Instead of using standard communication protocols for implementing reliable communication [20, 12], detection and recovery from errors in communication is part of the proposed fault tolerance scheme — simple special purpose hardware [23] is used to handle communication errors *without the need for message acknowledgements or for sending check bits with each message*. Furthermore, no “book-keeping” information (e.g., dependency vectors [20]) is sent with normal messages and there is no need to maintain extensive logs in parallel with normal operation, as required by message logging techniques [20, 11].

Several basic concepts and assumptions used in this paper are described in Section II. Section III presents definitions which are used in discussing distributed recovery. It also includes an overview of the basic approaches to application-transparent distributed recovery schemes and motivates the use of schemes based on coordinated recovery of interacting sets of processes. The error detection mechanisms used are described in Section IV. Section V includes a high-level abstract description of coordinated checkpointing and recovery of interacting sets and proves the correctness of the approach. The technique for identifying interacting sets of processes is described in Section VI, followed by a discussion of *synchronous* process-level checkpointing in Section VII [24]. *Asynchronous* process-level checkpointing is presented in Section VIII, followed by a description of recovery from errors using previously saved checkpoints in Section IX. Techniques for minimizing data movement during checkpointing by using an enhanced virtual memory system are outlined in Section X. An appendix contains a list of the various message types and their contents.

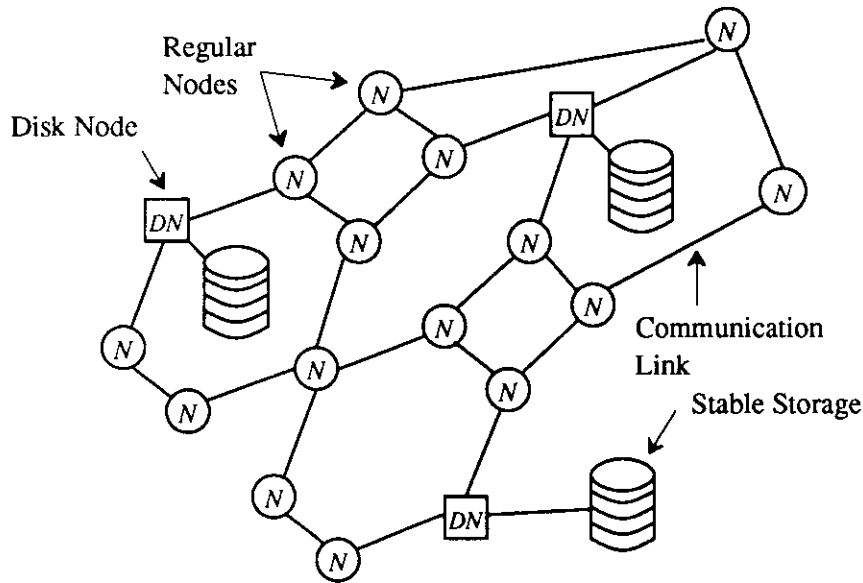


Figure 1: A multicomputer.

## II. Basic Concepts and Assumptions

The target systems is a multicomputer consisting of hundreds of nodes which communicate via messages over point-to-point links. Each node includes a processor, local memory, and a communication coprocessor. The nodes operate asynchronously and messages may have to pass through several intermediate nodes on their way to their destinations. The system is used for “general purpose” applications which have no hard real-time constraints. Errors can occur at any time as a result of *hardware* faults in the nodes or in the communication links.

Processes in the system communicate using *virtual channels* [2, 16], i.e., if processes on two nodes that are not immediate neighbors need to communicate, a logical circuit is set up from the source to the destination by placing appropriate entries in the routing tables of each intermediate node along the way. Once the path is set up, there is very little routing overhead for packets sent through the circuit and FIFO ordering of these messages is maintained. The connectivity of the system is high so that the probability of the system partitioning due to the failure of a node(s) is low enough so that it is reasonable for partitioning to cause a *crash*.

As discussed in Section IV, we assume that the nodes are self-checking and are guaranteed to signal an error to their neighbors immediately when they send incorrect output [22]. Hardware faults either cause a node to generate an error signal or cause an error in transmission.

The error recovery scheme is based on the existence of *stable storage* [13] where checkpoints can be safely maintained. Such stable storage may be implemented as mirrored disks. Some of the nodes in the

system, which we call *disk nodes*, are connected to such “reliable” disks. We assume that a failure of the disks themselves or of the disk nodes causes a *crash* (i.e., an unrecoverable error).

Processes are *checkpointed* periodically and rolled back to a previous checkpoint if an error is detected. The *state* of a process involved in checkpointing and recovery is the contents of all of the memory and registers used by the process. This includes some system tables, such as the list of all the virtual circuits currently established to and from the process. The state of a process changes as a result of local computation, the transmission of a packet, or the receipt of a packet.

In the proposed scheme, the unit of checkpointing and recovery is a set of interacting processes rather than individual processes, individual nodes, or the system as a whole [9,12]. Informally, an *interacting set of processes* is the set of processes which have communicated directly or indirectly since their last checkpoint. As will be discussed further in sections III and VI, if all messages in the system are “flushed” to their final destinations and no messages are lost [23,24], it is possible to partition the system into a collection of *disjoint* interacting sets of processes. Since there has been no communication between processes in different sets since their last checkpoint, a checkpoint of a process in one interacting set is consistent with both the checkpoint and the current state of another process which is in some other interacting set. Hence, different interacting sets may be checkpointed and recovered independently and a consistent global state (checkpoint) is always maintained.

Since each node can be time-shared between multiple processes, it may have to participate in multiple simultaneous checkpointing and recovery sessions. Hence, it is not advisable to implement checkpointing and recovery as part of the kernel. Instead, whenever checkpointing or recovery of a particular process is initiated, the kernel spawns a special *handler* process that performs the necessary operations. The handler can suspend the process, manipulate its state, and allow it to resume normal operation. In the rest of the paper we will often discuss the actions of participants in checkpointing and recovery sessions. These “participants” are really the handlers corresponding to the processes being checkpointed or recovered.

### III. Application-Transparent Distributed Error Recovery

Application-transparent error-recovery can be implemented using checkpointing and rollback: the states of processes are periodically saved (checkpointed) and previously saved process states are restored in order to recover from errors [15]. Stable storage (disk nodes) are used to save process states since it is assumed that all the data (state) in the local memory of a node will be lost if the node fails.

The interactions between processes must be taken into account by the checkpointing and rollback scheme. Specifically, if there is an error that forces the state of some of the processes to be restored to a

previously saved states, we must ensure that the restored states are *consistent* [5] with each other as well as with the states of all the other processes in the system. It must be the case that following recovery the execution of the system correspond to a possible execution that could have occurred in a fault-free system.

**Def. 1:** For a process  $p$ , let  $Buddies(p)$  be the set of processes in the system such that  $q \in Buddies(p)$  if, and only if, process  $p$  has either sent a message to  $q$  or received a message from  $q$  since the last checkpoint of  $p$ .

Note that in general  $q \in Buddies(p)$  does not necessarily imply that  $p \in Buddies(q)$ . For example, a message may have been sent by  $p$  but not yet received by  $q$ .

**Def. 2:** The *interacting set* with respect to process  $p_1$ ,  $Inter(p_1)$ , is the set of processes containing all processes  $q$  such that either (1)  $q \in Buddies(p_1)$ , (2)  $\exists$  a process  $p_2$ , such that  $p_2 \in Inter(p_1)$  and  $q \in Buddies(p_2)$ , or (3)  $q = p_1$ .

$Inter(p_1)$  is the set of processes that may be affected (directly or indirectly) by messages sent by  $p_1$  since the last checkpoint of  $p_1$  or which have produced messages which may have affected  $p_1$  directly or indirectly. Note that, in general,  $p_2 \in Inter(p_1)$  does not necessarily imply that  $p_1 \in Inter(p_2)$ .

**Def. 3:** Two processes,  $p$  and  $q$ , where  $q \in Buddies(p)$ , are said to be *buddy-consistent* if, and only if, their states reflect the same number and content of direct message exchanges between  $p$  and  $q$ .

Note that if  $q \in Buddies(p)$  and  $p$  and  $q$  are *buddy-consistent*, then it must be the case that  $p \in Buddies(q)$ . The definition implies that no messages between  $p$  and  $q$  are either lost or duplicated.

**Def. 4:** A set of processes,  $CPS$ , is said to be *consistent*, if, and only if, the following conditions hold: (1) if  $p \in CPS$  then  $q \in Inter(p)$  implies  $q \in CPS$ , (2) for every pair of processes in  $CPS$ ,  $p \in CPS$  and  $q \in CPS$ , one of the following three conditions holds: (2.a)  $q \notin Inter(p)$  and  $p \notin Inter(q)$ , (2.b)  $p$  and  $q$  are *buddy-consistent*, or (2.c)  $\exists$  a sequence of processes  $(p_1, p_2, \dots, p_m)$ , where  $\forall i, 1 \leq i \leq m, p_i \in CPS$ , such that  $\{p, p_1\}$  as well as  $\{p_m, q\}$  are *buddy-consistent* and  $\forall j, 1 \leq j < m, \{p_j, p_{j+1}\}$  are *buddy-consistent*.

If the system is fault free and all messages in transit throughout the system are flushed to their final destinations, all the processes in the system form a consistent process set (CPS). Note that a single process,  $p$ , for which  $\{p\} = Inter(p)$  is a consistent process set.

**Def. 5:** Two processes are said to be *consistent* if, and only if, there exists a consistent process set of which both are members.

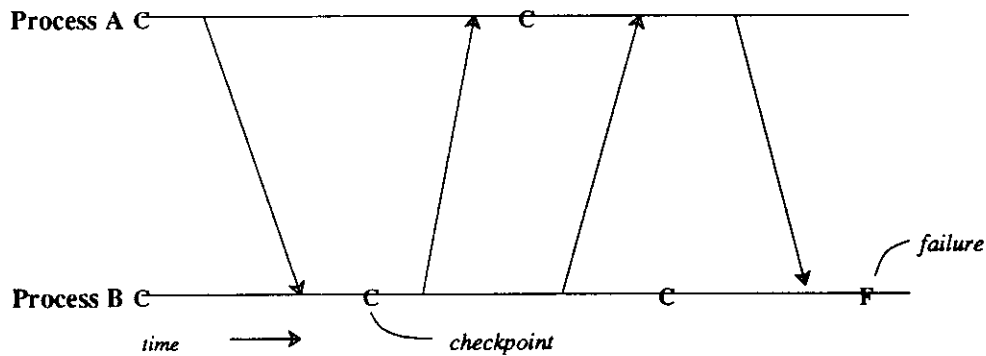
**Def. 6:** A set of processes,  $DCPS$ , is *dynamically consistent* if, and only if, the following holds: if the



execution of all processes in *DCPS* is blocked and all messages are flushed to their final destinations, all the members of *DCPS* form a consistent process set.

**Def. 7:** Two processes are said to be *dynamically consistent* if, and only if, there exists a dynamically consistent process set of which both are members.

If processes are checkpointed independently and rolled back independently when an error is detected, the state of the recovered process may not reflect messages that have been sent to it by other processes or the fact that it has sent messages to other processes before it failed. Thus, following recovery of a process, the set of all the processes in the system may no longer be dynamically consistent. This can lead to incorrect execution and erroneous results.



**Figure 2:** The *domino effect*. Process B fails and is rolled back to its latest checkpoint. Since B will later expect a message from A, A must also be rolled back, requiring B to roll back to a previous checkpoint. In this case, the entire system must roll back to its initial state.

The goal of distributed error recovery schemes is to restore the system to a state where all processes in the system form a dynamically-consistent process set. This can be accomplished by keeping multiple checkpoints of each process and tracking dependencies between each process  $p$ , and processes in  $Buddies(p)$ . If process  $p$  is rolled back, any process  $q \in Buddies(p)$  where  $\{p, q\}$  are not *buddy-consistent* must also be rolled back. This procedure is applied recursively to all the members of  $Inter(p)$ , and may cause multiple rollbacks of a single process to successively earlier checkpoints. In the worst case, this *domino effect* can cause all the processes in the system to roll back to their initial states (see Figure 2), thus losing all computations done so far [15].

There are two basic approaches to avoiding the domino effect in distributed checkpointing and rollback schemes:

I) *Message logging*, where messages as well as process states are saved in order to allow the state of a restored process to be “adjusted” so that it is consistent with other processes in the system [4, 11, 19, 20, 21].

II) *Coordinated checkpointing*, where processes are not checkpointed independently but are, instead, checkpointed in a coordinated way with some or all of the other processes in the system such that if recovery is necessary the restored states are guaranteed to be consistent [1, 12, 23, 24].

In the next two subsections we discuss these two approaches and motivate the choice of the second approach for error recovery in high-performance multicomputers.

#### A. *Message Logging*

With error recovery techniques based on message logging, process states are checkpointed and recovered independently. In addition to saving process states, interprocess messages are also saved (logged). When a process state is restored from a previous checkpoint, messages are “played back” to the process in order to bring it to a state that is consistent with the rest of the system [11, 20, 21].

A major problem with the use of message logging techniques for high-performance multicomputers is the amount of data that may be transmitted between processes (and thus will have to be logged) in applications that exploit fine-grain parallelism. Specifically, it is expected that some applications may require each process to send a short message of, say, eight bytes every 100 instructions [6, 10]. Consider a system with 512 20 MIPS processors. For this communication-intensive application, each one of these processors will send 200,000 messages (1.6 mega-bytes) per second. Thus, approximately 820 Mbytes per second are transmitted in messages throughout the system. Logging this much information to disk is a major problem since, given modern disks with a bandwidth of 4 Mbytes per second, more than 200 disk drives will have to be dedicated to logging messages. This problem is exacerbated by the use of faster processors.

An additional problem with message logging techniques is that they require bookkeeping information (dependency vectors, sequence numbers, etc) to be transmitted with each message, thus increasing the load on the communication network and making the send and receive operations, which have to manipulate this bookkeeping information, more complex. This information must also be logged to disk, further decreasing the network bandwidth available to the application and increasing the number of disks required to support message logging techniques.

There are techniques that allow message logging to be used without logging each message to stable storage [21]. However, with these techniques messages are temporarily logged in local memory, quickly filling up the local memory with the volatile message logs, thus interfering with normal processing. The need to log messages, even if just temporarily in local memory, increases local memory bandwidth requirement and may slow down processing due to conflicts in accessing local memory. In addition, these techniques require logging at least some sequence numbers to stable storage. Furthermore, as with

most message logging techniques, there is extra complexity, performance overhead, and network bandwidth overhead for keeping track of dependencies. A technique for simplifying and speeding up dependency tracking has been proposed in [19]. Unfortunately, there is a high cost associated with this technique during normal operation since it requires recomputing a past state of a process in order to update the checkpoint on stable storage.

One of the disadvantages of message logging techniques is that they require the application processes to be *deterministic*, i.e., given a process state and a sequence of inputs (message log), the process must generate the same outputs [20]. On the other hand, techniques based on coordinated checkpointing do not place any such restrictions on the behavior of application processes [12, 23].

Finally, message logging schemes assume reliable message transmission and thus require messages to be acknowledged and check bits to be transmitted with each message. As will be discussed in Section IV and Section VII this overhead may be avoided if the second approach to application-transparent error recovery is used.

### *B. Coordinated Checkpointing*

Barigazzi and Strigini [1] have proposed an error recovery procedure for multicomputers that involves periodic saving of the state of each process by storing it both on the node where it is executing and on another backup node. The critical feature of this procedure is that all interacting processes are checkpointed together, so that their checkpointed states are always consistent with each other. Therefore, the *domino effect* cannot occur and it is sufficient to store only one "generation" of checkpoints. The scheme presented in this paper uses this idea of checkpointing and recovering dynamically changing sets of interacting processes.

With the recovery scheme described in [1] a large percentage of the memory is used for backups rather than for active processes. The resulting increased paging activity leads to increases in the average memory access time and the load on the communication links. This load is also increased by the required acknowledgement of each message and transmission of redundant bits for error detection. The communication protocols, which are used to assure that the message "send" and "receive" operations are atomic, require additional memory and processing resources for the kernel. Thus, performance is significantly reduced relative to an identical system where no error recovery is implemented. The scheme proposed in this paper eliminates the requirements for atomic message transmission and provides the ability to save the checkpoints on disk, where they need not have a detrimental effect on system performance.

The idea of checkpointing and recovering interacting sets of processes is extended in [23] to

checkpointing and recovering the entire system (global checkpoints). That scheme does not have the disadvantages discussed above of the scheme in [1]. The problem with the global checkpointing technique is that checkpointing is expensive since it requires saving the state of the entire system. Thus, for performance reasons, the time between checkpoints is relatively long (possibly tens of minutes). Hence, the system can only be used for "batch applications," such as large numerical computations, where the possibility of losing minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [23]).

The global checkpointing technique has been extended to perform checkpointing and recovery of sets of interacting processes rather than of the entire system [12, 24]. This extension (see sections VI and VII) reduces the disruption to normal operation since it usually involves only a subset of the processes instead of the entire system. However, as in [23], these improved coordinated recovery schemes still involve *synchronous* checkpoint sessions, where all members of the interacting set being checkpointed suspend normal execution for the duration of the checkpoint session. In the worst case, all processes in the system may be in a single interacting set and all normal computation in the system has to be suspended for the duration of the checkpoint session. In this paper we further enhance the coordinated checkpointing technique by using *local volatile* checkpoints on each node to make the checkpoint session *asynchronous* with respect to normal processing. This new technique minimizes (nearly eliminates) the time during which normal process execution is suspended.

#### IV. Error Detection

As previously discussed, errors in the system may be a result of node failures or failures in the communication links. We assume that the nodes are self-checking and produce an error indication whenever their outputs are incorrect [17, 22].

In most systems, errors in message transmission are detected by including with each message *check bits*, which the receiver uses to determine whether the contents of a message has been corrupted. Lost messages are detected by protocols that involve acknowledging each messages as well as transmission of sequence numbers with each message [25]. The disadvantage of these techniques is that they involve transmission of redundant bits and thus "waste" communication bandwidth. Since the probability of an error in transmission is low, it is wasteful to check the validity of each message or packet independently. Instead, as proposed in [23], each node has two special purpose registers for error detection associated with each of its ports. One of these registers contains the CRC (Cyclic Redundancy Check) check bits for all the packets that have been sent from the port. The other register contains the CRC check bits for all packets received. These special purpose registers are linear feedback shift registers (LFSRs) and their

contents are updated in parallel with the transmission of each packet [8].

In order to check the validity of all the packets transmitted through a particular link, each node sends to its neighbor the contents of the LFSR used for outgoing packets. The neighbor can then compare the value it receives with the value in its LFSR for *incoming* packets and signal an error if it finds a mismatch. If packet switching is used, all the links in the system must be checked in this way before committing to a new checkpoint. Otherwise, the state of a node corrupted by an erroneous message may be checkpointed and later used for recovery. With virtual circuits, LFSRs at each node are used to accumulate signatures of the packets transmitted through each incoming and outgoing virtual circuit. Communication between processes in the interacting set can be checked, without checking all the links in the system, by performing “end-to-end” checks on all the virtual circuits between processes in that set.

The packets used to coordinate the creation of checkpoints and for error recovery must be verified before they are used. Hence, for these packets, an error detecting code is used and redundant bits are transmitted with the packet. Thus, there are two types of packets in the system: normal packets that do not include any information for error detection, and special control packets, called *fail-safe* packets, that are used only for transmitting information between handlers and which include a sufficient number of redundant bits to detect likely errors in transmission. The *fail-safe* packets are either error-free or the error is easily detectable by the receiving node.

## V. Using Interacting Sets for Checkpointing and Recovery

The proposed error recovery scheme is based on checkpointing consistent states of interacting sets of processes. More precisely, the following procedure is performed:

*Procedure Chkp:*

- 1) When a process  $p$  is checkpointed, all processes in  $Inter(p)$  (Def. 2) are also marked for checkpointing.
- 2) All messages sent by processes in  $Inter(p)$  are flushed to their final destinations.
- 3) The actual checkpointing is then performed.

**Lemma 1:** Assuming that the system is fault-free, the set of processes  $P = Inter(p)$ , which is checkpointed as described by Procedure *Chkp*, forms a *consistent process set*.

**Proof:** Consider  $q_1 \in P$  and  $q_2 \in Inter(q_1)$ . Based on condition (2) of Def. 2, this implies  $q_2 \in Inter(p)$  so condition (1) in Def. 4 holds.

Assume  $p_1 \in P$  and  $p_2 \in P$ . There are three cases to be considered: (1)  $p_2 \in Buddies(p_1)$ , (2)  $p_1 \in Buddies(p_2)$ , or (3)  $p_2 \notin Buddies(p_1)$  and  $p_1 \notin Buddies(p_2)$ . The case when both (1) and (2)

hold is covered by the discussion of either one.

If  $p_2 \in \text{Buddies}(p_1)$ , since the system is fault-free and messages from  $p_1$  and  $p_2$  have been flushed to their final destinations, it must be the case that the state of both processes reflect the same message exchanges between them and thus they are *buddy-consistent* (Def. 3). Similarly, condition 2.b in Def. 4 holds starting with  $p_1 \in \text{Buddies}(p_2)$ .

It remains to show that condition 2.c in Def. 4 holds when  $p_1 \notin \text{Buddies}(p_2)$  and  $p_2 \notin \text{Buddies}(p_1)$ . By condition (2) in Def. 2, there is a sequence of processes  $(x_1, x_2, \dots, x_m)$  where  $\forall i, 1 \leq i \leq m, x_i \in P$  such that  $x_1 \in \text{Buddies}(p)$ ,  $p_1 \in \text{Buddies}(x_m)$ , and  $\forall j, 1 \leq j < m, x_{j+1} \in \text{Buddies}(x_j)$ . Based on the above argument, the members of each adjacent pair of processes in the sequence  $(p, x_1, x_2, \dots, x_m, p_1)$  are *buddy-consistent*. Similarly, there is a sequence  $(p, y_1, y_2, \dots, y_k, p_2)$  where  $\forall i, 1 \leq i \leq k, y_i \in P$  and the members of each adjacent pair of processes are *buddy-consistent*. Hence, using the sequence  $(x_1, \dots, x_m, p, y_1, \dots, y_k, p_2)$ , condition 2.c in Def. 4 holds for  $p_1$  and  $p_2$ .  $\square$

**Lemma 2:** If every time a process  $p$  is checkpointed, its entire interacting set  $P = \text{Inter}(p)$  is also checkpointed as described in Procedure *Chkp*, then the set of all checkpointed processes (all the processes in the system),  $W$ , forms a consistent process set.

**Proof:** Since  $W$  includes all the processes in the system, condition (1) in Def. 4 is satisfied. Consider two processes in the system checkpoint  $p_1$  and  $p_2$ . If both processes were checkpointed together, then, by Lemma 1, they satisfy condition (2) in Def. 4. Assume  $p_1 \in P$  and  $p_2 \notin P$ . If  $p_2 \in \text{Inter}(p_1)$ , then, based on Def. 2,  $p_2 \in \text{Inter}(p)$ . Since this contradicts  $p_2 \notin P$ , it must be the case that  $p_2 \notin \text{Inter}(p_1)$ . Using the same argument, reversing the roles of  $p_1$  and  $p_2$ , it must also be the case that  $p_1 \notin \text{Inter}(p_2)$ . Hence, condition (2.a) in Def. 4 holds.  $\square$

Based on Lemma 2, when an error occurs, we can restore a consistent system state by discarding all messages in transit and restoring all processes to their checkpointed states. However, in order to reduce the impact of recovery on system operation, it is desirable to reduce the number of processes that have to be rolled back. When an error is detected, all the processes that *could have been* affected by the error are identified. The sets of processes that have interacted with the affected processes since their last checkpoint are determined, and the states of all these processes are rolled back to that last checkpoint. The rest of this section shows that if this procedure is followed, following recovery, all the processes in the system form a *dynamically consistent process set*.

The recovery operation proceeds as follows:

*Procedure Rback:*

- 1) A process  $p$  is marked for rollback.
- 2) Any process  $q$ , s.t.  $p \in \text{Buddies}(q)$  is marked for rollback.
- 3) For each  $q$ , all processes  $x \in \text{Inter}(q)$  are marked for rollback.
- 4) All messages in transit sent from processes marked for rollback are discarded from the network.
- 5) All processes marked for rollback are restored to their last checkpoint and normal operation is resumed.

**Def. 8:** If process  $p$  is marked for rollback, we denote by  $\text{Recov}(p)$  the entire set of processes which must be rolled back, as determined by Procedure *Rback*.

**Lemma 3:** If the most recent checkpoints of all the processes in the system form a consistent process set and Procedure *Rback* is performed for some process  $p$ , all the processes *running* on the system,  $R$ , (as opposed to their checkpointed states) form a *dynamically consistent process set*.

**Proof:** Since  $R$  includes all the processes in the system, condition (1) in Def. 4 is satisfied. Consider two processes  $p_1$  and  $p_2$ . If both processes are in  $\text{Recov}(p)$ , they are recovered together from their last checkpoint. Hence, by assumption, both processes are in the same consistent process set. Thus, condition (2) in Def. 4 holds.

Consider the case when  $p_1 \notin \text{Recov}(p)$  and  $p_2 \notin \text{Recov}(p)$ . Assuming that the system was fault-free prior to the rollback, all processes in the system were in a dynamically consistent process set. Hence, if all messages were flushed to their destinations, one of the clauses in condition (2) of Def. 4 must have held for  $p_1$  and  $p_2$ . The rollback, that did not involve  $p_1$  and  $p_2$ , could not change the situation if condition (2.a) or (2.b) of Def. 4 held. If condition (2.c) of Def. 4 held, the sequence  $(p_1, x_1, x_2, \dots, x_m, p_2)$  where consecutive processes are *buddy-consistent* could not have included any process in  $\text{Recov}(p)$ . If the sequence did include a process in  $\text{Recov}(p)$ , then both  $p_1$  and  $p_2$  would be in  $\text{Recov}(p)$ . Thus, condition (2.c) in Def. 4 must still hold for  $p_1$  and  $p_2$ .

Assume  $p_1 \in \text{Recov}(p)$  and  $p_2 \notin \text{Recov}(p)$ .

Consider the possibility that  $p_2 \in \text{Inter}(p_1)$ . If  $p_1$  was added to  $\text{Recov}(p)$  in Step (2) of Procedure *Rback*,  $p_2 \in \text{Inter}(p_1)$  implies  $p_2 \in \text{Recov}(p)$  — a contradiction. If  $p_1$  was added to  $\text{Recov}(p)$  in Step (1) or (3) of Procedure *Rback*,  $\exists q \in \text{Recov}(p)$  s.t.  $p_1 \in \text{Inter}(q)$ . In this case  $p_2 \in \text{Inter}(p_1)$  implies  $p_2 \in \text{Inter}(q)$  which implies  $p_2 \in \text{Recov}(p)$  — a contradiction. Hence, it must be the case that  $p_2 \notin \text{Inter}(p_1)$ .

Since  $p_2 \notin \text{Inter}(p_1)$ , if  $p_1 \in \text{Inter}(p_2)$ , then condition (2.a) in Def. 4 holds, and the proof is complete.

It remains to consider the case  $p_1 \in Inter(p_2)$ . Based on Def. 2,  $\exists$  a sequence of processes  $(x_1, x_2, \dots, x_m)$  where  $x_1 \in Buddies(p_2)$ ,  $p_1 \in Buddies(x_m)$ , and  $x_{j+1} \in Buddies(x_j) \forall j \in [1, m)$ . At some point in this sequence the “boundary” between processes outside  $Recov(p)$  and processes in  $Recov(p)$  is crossed, i.e.,  $\exists z \in Recov(p) \ y \notin Recov(p)$ , where  $z \in Buddies(y)$ . Since  $y \notin Recov(p)$ , it must be the case that  $y \notin Buddies(z)$ . Hence, the only interaction between  $y$  and  $z$  is a message in transit from  $y$  to  $z$ . If all messages are flushed to their destinations, the message from  $y$  will arrive at  $z$  and  $\{y, z\}$  will become *buddy-consistent*. At this point, condition (2.c) in Def. 4 will hold.  $\square$

**Theorem 1:** If Procedure *Chkp* is used for checkpointing and Procedure *Rback* is used for recovery, all processes in the system will form a *dynamically consistent process set* following recovery.

**Proof:** Directly from Lemma 2 and Lemma 3.  $\square$

The rest of this paper can be viewed as describing techniques for efficient implementation of the procedures described above.

## VI. Identifying Dynamic Interacting Sets

In the previous section we showed that checkpointing and recovering interacting sets of processes can be the basis of an error recovery scheme. It remains to show how these high-level abstract procedures can be translated into a practical scheme. Checkpointing and recovery sessions require coordination. This is accomplished by a *coordinator* handler that is dynamically determined as part of each session. The mechanism for identifying the participants in checkpointing and recovery sessions and for selecting coordinators will be described in this section.

An interacting set of processes forms a *communication graph* where there is a vertex for each process and each arc indicates that communication has taken place between the two processes it connects. The communication graph can be transformed into a *communication tree* by designating one of the vertices as the “root process” or *coordinator*. All vertices which have arcs connected to the root (“children” of the root) are called *first-level processes*. Processes/vertices which have no children are called *leaves*. The communication tree is the fundamental unit around which our algorithms are structured.

When checkpointing or recovery is initiated, the kernel spawns a *handler* process that performs the necessary operations. A handler initiated as a direct result of a “checkpointing timer” triggering or an error being detected begins its operation assuming that it will be the coordinator of a checkpointing or recovery session. In order to enable such a handler to form a communication tree, the system (hardware and/or software) must maintain, for each process  $p$ , *dynamic communication information* which is the list



of processes with which there has been *direct* communication since the last checkpointing session (i.e., *Buddies(p)*) [1]. This list is called a *first-level list* since, if the process becomes a coordinator (and root of a communication tree), the processes on this list are the *first-level processes* mentioned above.

The coordinator initiates formation of a communication tree by sending *CHECKPOINT* or *ROLLBACK* messages to all the processes on its first-level list. These processes are then placed in either a “checkpointing” or “recovering” state, handler processes are spawned for them, the handlers send *CHECKPOINT/ROLLBACK* messages to all their first-level processes (except for the parent process), and so on. A process that is already part of the tree informs the sender that it will not be its child. A process is a *leaf process* of a communication tree if it has communicated only with the process that sent it a *CHECKPOINT/ROLLBACK* message, or processes that are already part of the communication tree. Each leaf process informs its parent that it is its child and that it is a leaf. Each non-leaf process waits for confirmations/denials from the roots of all its subtrees and then sends a confirmation acknowledgement to its parent. This level-by-level process continues back up to the root process. When the final acknowledgement is received by the root process, the communication tree is complete - the interacting set has been found. All stages of the algorithms described later proceed in this step-wise fashion.

It is possible for several processes within an interacting set to initiate checkpointing and/or recovery sessions simultaneously. Due to the stepwise confirmation/denial process it is possible to create a correct and consistent communication tree by “disassembling” all but one of the subtrees and incorporating their members in the single “winning” tree. This is described in the next section where synchronous process-level checkpointing is discussed.

## VII. Synchronous Process-Level Checkpointing

In this section we summarize the checkpointing algorithm presented in [24]. Checkpointing is triggered by a “checkpointing timer,” which causes the kernel to spawn a *handler* process. This handler begins its operation assuming that it will *coordinate* a checkpointing session for the interacting set of the process with which the timer is associated. In synchronous checkpointing, once a process becomes involved in a checkpoint session it does not execute again until its checkpoint has been committed to disk. Checkpointing each process involves the following basic steps:

1. The process is halted. A *handler* process, that will participate in the checkpointing session on behalf of the halted process, is initiated.
2. The portion of the process state that is not in memory (e.g. contents of registers) is stored in dedicated buffer area in local memory.

3. The process state in memory (or, as discussed in Section X, just the modified pages) is sent to stable storage.
4. The interacting set/checkpoint tree is found. The communication channels between processes in the interacting set are *flushed* and any packets flushed from incoming channels are sent to stable storage as part of the checkpoint state.
5. Once all the checkpoint states, including flushed packets, of all the processes in the interacting set have been written to stable store, those states are committed and the handlers terminate. Processes then resume normal operation.

If the checkpointing session is triggered by the timer of process X, the *checkpoint coordinator*,  $CC_X$ , is responsible for sending process X's state to disk; finding the current interacting set (*checkpoint tree*) by sending CHECKPOINT messages (markers with error detection bits) to all *first-level processes* Y (as defined in Section VI); and checking for communication errors on all channels from processes Y to process X (as described in Section IV).

Any process, Y, receiving a CHECKPOINT message will be suspended and *checkpoint handlers* ( $CH_Y$ ) will be started up in its place. This handler is responsible for copying Y's state and sending it to disk; sending CHECKPOINT messages to all first-level processes *except* for the coordinator/parent; and checking for any errors on all of Y's incoming channels (virtual circuits). This continues until *leaf processes* are found — processes which have communicated only with the sender of the CHECKPOINT message received OR which have communicated only with processes already part of the tree. The checkpoint handler for each leaf process then sends a CH\_ACK message (marker with error detection bits) to its "parent". Any handler receiving a second, or more, CHECKPOINT message (i.e. already has a "parent" in the checkpoint tree) receives and sends a marker and checks for communication errors but denies child status of that handler.

Once a parent of a leaf process receives CH\_ACK messages from all its children, a CH\_ACK message is sent to its parent, and so on. When  $CC_X$  receives CH\_ACK messages from all processes to which it sent CHECKPOINT messages, the entire interacting set has been found. Thus, CHECKPOINT and CH\_ACK messages serve the purpose of *flushing* messages to their destinations, carrying error detection bits from one end of a channel to the other for comparison, and finding the interacting set.

Once  $CC_X$  receives CH\_ACK messages from all its first-level processes, it sends a CH\_FOUND message down the tree, thus notifying the handlers that all messages have been flushed to their destinations and message queues can now be sent to disk. Once the message queues have been saved on disk, CH\_DONE messages are sent from the leaf processes to their parents, and so on. When  $CC_X$

receives CH\_DONE messages from all its children, it “knows” that the process states of all the processes in the tree have been written to disk.

$CC_X$  commits all processes to the new checkpoint by directing its disk node to commit to the new checkpoint of X and destroy X’s previous checkpoint. When the disk node acknowledges this operation,  $CC_X$  sends CH\_COMMIT messages down the tree. CH\_RESUME messages are sent up the tree after each handler commits its process’ state to disk. Each handler which sends a CH\_RESUME flags its process as “runnable” and terminates.  $CC_X$  terminates last. Processes cannot participate in or initiate new checkpointing sessions until they receive the CH\_RESUME message. The *first-level list* for a process (which contains the dynamic communication information) is cleared by its handler upon receiving the CH\_RESUME message.

#### A. Concurrent Invocations of the Algorithms

As mentioned in Section VI, it is possible for several processes within an interacting set to initiate checkpointing and/or recovery sessions “simultaneously”. To solve this problem a single coordinator is deterministically chosen to coordinate the session. To this end, CHECKPOINT and ROLLBACK messages contain checkpoint/recovery coordinator IDs and all handlers locally store their current coordinator ID. Since there is a total ordering of node and process identifiers, a process receiving CHECKPOINT or ROLLBACK messages originating from different coordinators can pick the coordinator with the “largest” ID. However, ROLLBACK messages always win over CHECKPOINT messages regardless of the coordinator ID. Specifically, a ROLLBACK message will cause a checkpointing handler to immediately stop all activities related to checkpointing and join the recovery session. Since the first-level list is not cleared until the process has completed all phases of a checkpointing session, the recovery session is guaranteed to include all the handlers in the ongoing checkpointing session that may otherwise be waiting indefinitely for checkpointing-related responses. If a change in coordinator or session type is made then the handler propagates this decision to its first-level processes by resending CHECKPOINT/ROLLBACK messages with the new coordinator ID. Eventually the winning coordinator will “flush out” all remnants of the losing session.

### VIII. Asynchronous Process-Level Checkpointing

The key feature of asynchronous checkpointing is that interruption of normal processing is minimized: the normal execution of a participating process is suspended only as long as it takes to copy its state in local memory (see Section X for further optimizations). Each process in the interacting set is “checkpointed” in *local volatile* memory and then resumes normal operation while the volatile

checkpoints are copied to stable storage. Until Section VIII.4 we assume that a process will participate in only one checkpoint at a time - that is, a process may not start or join a second checkpoint session until the first one is committed to disk. Checkpointing each process involves the following basic steps:

1. The process is halted. A *handler* process, that will participate in the checkpointing session on behalf of the halted process, is initiated.
2. The portion of the process state that is not in memory (e.g. contents of registers) is stored in dedicated buffer area in local memory.
3. The process state in memory (or, as discussed in Section X, just the modified pages) is copied to the dedicated buffer area for later transmission to stable storage.
4. The communication channels to/from neighbors are *flushed* and any packets flushed from incoming channels are included in the local volatile checkpoint in local memory.
5. The process resumes normal operation.
6. The *handler* continues to participate in the checkpointing session. Specifically, in identifying the entire interacting set and committing the coordinated checkpoint to stable storage.

Implementation of the above steps is straightforward if the first-level list of each process is fixed, i.e., if the set of processes that each process interacts with between checkpoints is static (predetermined). Under these conditions, as described in the next subsection, the interacting sets are *static* and the algorithms for synchronous process-level checkpointing (Section VII) can be used here with little modification.

In most systems the interacting sets will be different for each checkpointing session. Specifically, an interacting set  $S$  containing some process  $P$ , grows with time since  $P$  was last checkpointed as more processes that are not members of  $S$  communicate with processes that are in  $S$ . A critical step in synchronous process-level checkpointing is identification of the entire interacting set so that *all* messages between members of the set are flushed to their destinations [24] (see Section VII). As will be shown in Subsection VIII.2, this problem is more difficult to deal with when asynchronous checkpointing is used since members of the interacting set resume normal processing (i.e. sending messages) *before* the entire interacting set is identified.

#### A. *Asynchronous Checkpointing of Static Interacting Sets*

If it is statically determined which processes are in an interacting set, creating a checkpoint tree is a simple operation. When the *handler* process receives a CHECKPOINT message, it sends CHECKPOINT messages to all first-level processes, sends a CH\_ACK message to its parent handler (unless it is the *root*

*handler*), and then waits for CH\_ACK messages to arrive from each first-level process. Since all the immediate neighbors (the first-level processes) are known (predetermined), once all the expected CH\_ACK messages arrive, it is known that *all* messages on both incoming and outgoing virtual circuits have been flushed to their destinations. At this point, the volatile checkpoint state is complete and consistent with the checkpoint state of other processes in the interacting set for the current checkpointing session.

Once a handler receives the CH\_ACK message from all its first-level processes, it completes the volatile checkpoint in local memory and *the process can resume normal operation*. The handler is now free to send, or finish sending, its process' volatile checkpoint to stable store. When the handler of a leaf process finishes this task, it sends a CH\_DONE message to its parent handler. A handler of non-leaf processes sends the CH\_DONE message to its parent only after it finishes sending its process' checkpoint to stable store *and* it receives CH\_DONE messages from all of its first-level processes. After sending the CH\_DONE message, a handler waits for a CH\_RESUME message. When the CH\_RESUME arrives, the handler directs its disk node to commit to the most recent checkpoint and forwards the CH\_RESUME message to all its first-level processes. The *root handler* sends the CH\_RESUME message as soon as it receives CH\_DONE messages from all its first-level processes and confirms that its disk node has committed to its most recent checkpoint. In final phase of the checkpointing session the leaf handlers send a CH\_RESUME\_ACK to their parents and are ready to participate in a new checkpointing session.

The key difference between the algorithm above and the synchronous algorithm [24] is that the algorithm presented here allows processes can begin normal execution long before the checkpointing session is complete (see Section X). Since the interacting set is static, the algorithm can avoid one round of messages through the tree which are used in [24] to determine which processes are members of the tree (in [24] CH\_ACK messages are forwarded from the leaf nodes to the root, and CH\_COMMIT messages are sent back to the leaf nodes before the leaf nodes can send the CH\_DONE messages to the root).

The rounds of messages can be related to the six basic steps described above, as follows: The CHECKPOINT messages initiate Step 1. From this point on, checkpointing activities are performed by the handlers. Steps 2 and 3 are initiated before receiving CH\_ACK messages. Step 4 can be completed once all CH\_ACK messages are received. Once Steps 2-4 are complete, the process resumes normal operation (Step 5). The remaining rounds of messages are all part of Step 6 — completion of all activities related to the checkpointing session by the checkpointing handler.

### B. Asynchronous Checkpointing of Dynamic Interacting Sets

In this environment the checkpointing scheme is more complex since new processes may join the interacting set in the middle of a checkpointing session (a process  $P_1$ , which is not a member of an interacting set  $S$ , joins  $S$  by communicating with some process  $P_2$  which is a member of  $S$ ). In this section the problems caused by dynamically changing interacting sets are described and solutions are proposed.

An important difference between the asynchronous algorithm for dynamic interacting sets and both the synchronous algorithm [24] and the asynchronous algorithms for static interacting sets (Subsection VIII.1) is the use of CH\_ACK messages. Just as in the algorithm for static interacting sets, the CH\_ACK message is sent immediately to the parent (the sender of the CHECKPOINT message) to allow that process to take a volatile checkpoint and resume normal operation without further delays. However, since the interacting set is dynamic, receiving the CH\_ACK message from all known first-level processes is not sufficient to ensure that all incoming circuits are flushed [24]. Hence, a new message type must be used to notify the coordinator when the entire interacting set is found. The SUBTREE message is used for this task. When a node which receives a CHECKPOINT message determines that it is a leaf node, it responds to its parent with a SUBTREE message as well as a CH\_ACK message. SUBTREE messages are forwarded step-by-step to the root (the coordinator). When the root receives SUBTREE messages from all its children, it “knows” that the entire interacting set has been found.

With asynchronous checkpointing a *checkpointing session* begins when the coordinator decides to take a checkpoint and ends when the checkpoints of all members of the interacting set are committed in stable storage. The interacting set may change during a checkpointing session under two conditions:

Case 1 — A process  $P_i$  in  $S$  sends a message  $M$  to a process  $P_o$  which is possibly outside  $S$ .

Case 2 — A process  $P_o$  which is possibly outside  $S$  sends a message  $M$  to a process  $P_i$  which is in  $S$ .

The statement “process  $P_i$  is in  $S$ ” means that a volatile checkpoint of  $P_i$  has been taken but  $P_i$  has not yet been informed that its checkpoint is committed in stable storage. When  $P_i$  sends or receives the message  $M$  to/from  $P_o$  it does not know whether  $P_o$  is in the interacting set. All it knows is that there has not been any prior *direct* communication between  $P_i$  and  $P_o$  since the last checkpointing of  $P_i$ .

Consider Case 1. If it turns out that  $P_o$  is not a member of  $S$ , the fact that a message was sent to it after the volatile checkpoint of  $P_i$  has no effect on the checkpointing session of  $S$ . In this case, the checkpointing session can safely end by committing to the state of  $P_i$  prior to sending  $M$ . On the other hand, if  $P_o$  turns out to be a member of  $S$ , and is informed of its participation in this checkpointing session *after* receiving  $M$ , the checkpoint of  $P_o$  will not be consistent with the state of  $P_i$  prior to sending

$M$  (i.e., following recovery we would end up with  $P_i$  in a state prior to sending  $M$  and  $P_o$  in the state after receiving  $M$  resulting, effectively, in duplicate transmission of  $M$ ). In this case we would like to “adjust” the checkpoint of  $P_i$  so that it corresponds to the state of  $P_i$  *after* sending  $M$ .

Similarly, consider Case 2 above. If it turns out that  $P_o$  is not a member of  $S$ , the fact that  $M$  was received from it after the volatile checkpoint of  $P_i$  was taken has no effect on the checkpointing session of  $S$ . The checkpoint of  $P_i$  should not contain  $M$  or state changes to  $P_i$  as a result of  $M$ . On the other hand, if  $P_o$  turns out to be a member of  $S$ , and is informed of its participation in this checkpointing session *after* transmitting  $M$ , the checkpoint of  $P_o$  will not be consistent with the state of  $P_i$  prior to receiving  $M$  (i.e., following recovery we would end up with  $P_i$  in a state prior to receiving  $M$  and  $P_o$  in the state after transmitting  $M$  resulting, effectively, in the loss of message  $M$ ). In this case, we would like to “adjust” the checkpoint of  $P_i$  so that it corresponds to the state of  $P_i$  *after* receiving  $M$ .

Both of the problems above can be solved by preventing processes from restarting normal operation until the entire interacting set is defined. This means that, after the volatile checkpoint is taken, a process  $P_i$  will not run, send messages, or receive messages until it receives a (CH\_COMMIT) message (see Section VII) which is broadcast from the checkpoint coordinator to all members of the interacting set. During the time that the process  $P_i$  is “frozen,” messages destined to it may arrive at its node. These messages are buffered and tagged with the identifiers of their senders. If the sender  $P_s$  is part of the interacting set, the channel from  $P_s$  to  $P_i$  is flushed immediately after  $P_s$  joins the checkpoint session and before  $P_i$  receives the CH\_COMMIT message (see Section VII). Flushing involves sending a “marker” [5] (CHECKPOINT or CH\_ACK message, see Section VII) from  $P_s$  to  $P_i$ . This marker causes all messages from  $P_s$  stored in the buffer since  $P_i$  joined the checkpointing session to be included with the checkpoint of  $P_i$ .

The solution of blocking processes until the interacting set is defined is simple to implement but may be undesirable since it involves temporary interruption of normal processing. Ideally, the system should allow processes that are taking part in a checkpointing session to continue normal processing immediately after taking a volatile checkpoint. This requires more complex *send* and *receive* operations during the interval in which the process would be blocked if the simple scheme above is used. Specifically, during this interval every message received by  $P_i$  from a process  $P_s$  that is not known (by  $P_i$ ) to be part of the interacting set, must be copied to a *holding buffer* in addition to being normally received by  $P_i$ . If it is later discovered that  $P_s$  is, in fact, part of the interacting set, messages from  $P_s$  that are in the holding buffer must be incorporated in the checkpoint of  $P_i$ . This solves one of the possible problems with dynamic interacting sets (Case 2 above).

The other part of the problem with dynamic interacting sets (Case 1 above) is more difficult to solve without introducing the possibility of blocking the process until the interacting set is identified. The problem is how to handle the situation where a process  $P_i$ , which is part of the interacting set, initiates communication (sends a message  $M$ ) with a process  $P_o$  which is possibly not part of the interacting set. Three possible solutions have been considered:

- A. Incorporate  $P_o$  in the interacting set, forcing it to checkpoint *before* receiving  $M$ . This can be done by  $P_i$  sending a CHECKPOINT message to  $P_o$  before sending  $M$ .
- B. Send the message  $M$  and, if  $P_o$  turns out to be part of the interacting set, “adjust” the state of  $P_i$  before completing the checkpointing session to reflect the fact that  $M$  has been sent.
- C. Block the sending of  $M$  to  $P_o$  but allow  $P_i$  to continue executing (assuming the *send* executed by  $P_i$  is a non-blocking asynchronous send).

Solution A is undesirable because  $P_i$  may have already sent a SUBTREE message to its parent informing it that its subtree is complete. This is a problem especially when  $P_o$  is actually outside of the interacting set (i.e. has not communicated with any process in the interacting set except through message  $M$ ). In such a case  $P_i$  has, by sending a CHECKPOINT message to  $P_o$ , actually expanded the interacting set by  $P_o$  and all processes with whom  $P_o$  has directly or indirectly communicated. Hence,  $P_i$  had falsely informed its parent that its subtree was complete and, if the checkpoint coordinator sends the CH\_COMMIT message before  $P_i$  can correct this misinformation then, if the checkpoint session is allowed to complete the checkpoints will be inconsistent. Also, in the worst case, the interacting set can continue to expand in this manner indefinitely.

Solution B suffers from a similar problem as solution A. In order to “adjust” the state of  $P_i$  to reflect the fact that  $M$  has been sent,  $P_i$ 's handler must suspend  $P_i$ , take a *new* volatile checkpoint of  $P_i$ , and resend CHECKPOINT messages as before, although the interacting set may be expanded by the processing activities that  $P_i$  has undergone since the previous volatile checkpoint was taken. Effectively,  $P_i$ 's checkpoint has been restarted and it can not be guaranteed with this policy that it, or some other process in the interacting set, will not restart again and hence, in the pathological case, the checkpoint may never finish.

In solution C the sending of  $M$  to  $P_o$  is blocked but  $P_i$  is allowed to continue executing, unless the send executed by  $P_i$  is a blocking send, in which case  $P_i$  must be blocked. If the send is non-blocking, then  $P_i$  can continue executing but any messages to  $P_o$  will not be sent. The blocked message(s) to  $P_o$  can be sent as soon as it has been determined whether  $P_o$  is part of the interacting set, i.e., when  $P_i$



receives either a CH\_COMMIT message from its parent or a CHECKPOINT message (with matching checkpoint coordinator ID) from  $P_o$ . In the latter case  $P_i$  must send a CH\_ACK message to  $P_o$  prior to sending any of the blocked messages. This solves the second part of the problem (case 1 above) with dynamic interacting sets in asynchronous checkpointing.

### C. Handling Coordinators With Different IDs

Since timers are used to initiate checkpoint sessions, it is possible for more than one process in a single interacting set to “simultaneously” start a checkpointing session. Hence, a process  $P_i$ , involved in a checkpoint  $S$  with coordinator  $B$ , may receive a CHECKPOINT message from process  $P_o$  with a different coordinator ID. This different coordinator may “win” or “lose” relative to  $P_i$ ’s current coordinator (as described in the Section VII) and  $P_o$  may possibly be outside or inside the interacting set. Hence, there are four cases that need to be handled:

1.  $P_o$  is *not* a first-level process of  $P_i$  and  $P_o$ ’s coordinator loses to  $P_i$ ’s coordinator.
2.  $P_o$  is *not* a first-level process of  $P_i$  and  $P_o$ ’s coordinator wins over  $P_i$ ’s coordinator.
3.  $P_o$  is a first-level process of  $P_i$  and  $P_o$ ’s coordinator loses to  $P_i$ ’s coordinator.
4.  $P_o$  is a first-level process of  $P_i$  and  $P_o$ ’s coordinator wins over  $P_i$ ’s coordinator.

For cases 1-2, the CHECKPOINT message from  $P_o$  is stored in the holding buffer that the handler for  $P_i$  will maintain for messages from  $P_o$ . Handling of this message is delayed until  $P_i$  receives either

- (a) a CHECKPOINT message from  $P_o$  with coordinator ID  $B$ . This implies that  $P_o$  is actually part of the interacting set and the previously sent CHECKPOINT message had a *losing* coordinator ID.
- (b) a CHECKPOINT message from some other, first-level process of  $P_i$ , with  $P_o$ ’s *winning* coordinator ID. In this case  $P_o$  is part of the interacting set and had previously sent a CHECKPOINT message with a winning coordinator ID.  $P_o$ ’s coordinator did not “flush out”  $P_i$ ’s coordinator earlier because  $P_o$  is not a first-level process of  $P_i$  and we do not want to expand the interacting set unnecessarily, as discussed in the previous subsection.
- (c) a CH\_COMMIT message from  $P_i$ ’s parent. In this case  $P_o$  is actually outside the interacting set.

In (a) and (b)  $P_o$  is allowed to join the checkpoint and the losing coordinator is “flushed out” as described in Section VII. In (a)  $P_i$  sends a CH\_ACK to  $P_o$  and removes the previous CHECKPOINT message from the holding buffer. In (b)  $P_i$  sends a CH\_ACK message (with  $P_o$ ’s coordinator ID) to  $P_o$  and sends CHECKPOINT messages with the new coordinator ID to its previous parent and children. In (c) it is possible to either reject or join  $P_o$ ’s checkpoint session. This choice will be discussed in the next section where we show how to involve processes in more than one checkpoint at a time.

For case 3 ( $P_o$ 's coordinator is the loser),  $P_o$ 's coordinator is "flushed out" by sending a CHECKPOINT message to  $P_o$  with  $P_i$ 's coordinator ID,  $B$ .

For case 4 ( $P_o$ 's coordinator is the winner),  $P_i$  becomes  $P_o$ 's child.  $P_i$  sends CHECKPOINT messages with the new coordinator ID to its previous parent and children.

#### D. Handling Multiple Checkpointing Sessions

In this subsection we relax the previously stated assumption that a process will not participate in more than one checkpointing session at a time.

A second checkpointing session may not be joined until the interacting set has been found for the first checkpoint session since it is only at that time that a checkpoint handler can be sure that the two checkpoints are actually for two *different* interacting sets. When a checkpoint coordinator has received SUBTREE messages from all its children or when a checkpoint handler has received a CH\_COMMIT message, then the handler/coordinator examines all holding buffers for outstanding CHECKPOINT messages with (losing) coordinator IDs. At this time the handler for some process  $P_i$  must reply to the sender of any such CHECKPOINT message(s),  $P_o$ .  $P_i$  has the choice of *rejecting* the checkpoint or *joining* the checkpoint as a leaf process. It should be noted that the checkpoint session headed by the losing coordinator ID has effectively been "on hold" as it cannot proceed to the *commit* phase until all CH\_ACK and SUBTREE messages have been received.

Once  $P_i$ 's interacting set has been found and  $P_o$  is not part of it,  $P_i$ 's handler can *reject*  $P_o$ 's checkpoint. At this time  $P_i$ 's handler can send a REJECT\_CHECKPOINT message to  $P_o$  and then complete its own checkpoint. The REJECT\_CHECKPOINT message is propagated by  $P_o$  to the rest of its checkpoint tree, erasing all traces of the checkpointing session.

An alternative course of action is for  $P_i$  to *join*  $P_o$ 's checkpoint as a leaf process by sending a CH\_ACK message to  $P_o$ .  $P_o$  will now be  $P_i$ 's parent for this checkpoint. Despite the fact that  $P_i$  has been executing while  $P_i$ 's handler has been performing checkpoint duties,  $P_i$  need not *take* another checkpoint (copy a new state in volatile memory) in order to join  $P_o$ 's checkpoint. This is because process  $P_i$  has not been allowed to expand the interacting set. Specifically, it has not been allowed to send messages to  $P_o$  (directly *or* indirectly). Therefore any messages received from  $P_o$  were sent regardless of any *post*-checkpoint state activities of  $P_i$ . Hence, the only *new* (changed) checkpoint state of  $P_i$  that needs to be sent to disk are all messages sent by  $P_o$  to  $P_i$  prior to  $P_o$  sending the CHECKPOINT message.

At this point  $P_i$ 's handler treats the second checkpoint just like any other except that it must ensure

that  $P_i$ 's first checkpoint is committed to disk *before*  $P_i$ 's second checkpoint. This is accomplished by simply not sending a CH\_DONE message to  $P_o$  until the first checkpoint has been committed (a CH\_RESUME is received).

## IX. Recovery

Recovery sessions are initiated when an error is detected. If there is a mismatch between the two signatures at the ends of a virtual circuit, a single interacting set needs to be rolled back — the set containing the two processes on either end of the virtual circuit. When an error in a node's outputs is detected by a neighboring node [22], recovery may involve more than one interacting set since all processes which were executing on the node and their interacting sets must be rolled back. If any messages were in transit on the node at the time of failure, recovery will be initiated when the signature mismatch is later detected.

### A. Recovery From Communication Errors

The algorithm for recovery from a communication error is similar in structure to the *synchronous* checkpointing algorithm, where recovery coordinators and handlers replace their checkpointing counterparts. When a signature mismatch is detected, the kernel initiates rollback by spawning a *recovery coordinator* process. A *recovery tree* is created by propagating ROLLBACK messages which are acknowledged by RE\_ACK(CHILD/NOT\_CHILD) messages just as CHECKPOINT and CH\_ACK messages are used. Unlike checkpointing, no signature comparisons are made, and all messages which are flushed to their destinations are discarded (this takes care of the *recovery livelock* problem discussed in [12]). For each process Y in the interacting set, the associated recovery handler  $RH_Y$  requests the process' state from the appropriate disk node after all expected RE\_ACK messages have arrived. When a handler receives its entire process state and associated message queue as well as RE\_DONE messages from all its children, it sends a RE\_DONE message to its parent. When the recovery coordinator receives its checkpointed state and RE\_DONE messages from all its children, it sends RE\_RESUME messages to its children, marks its associated process "runnable", and terminates. Upon receipt of a RE\_RESUME,  $RH_Y$  terminates and process Y resumes normal processing. No "commit" or "resume acknowledgement" phases are needed.

### B. Recovery From Node Failures

When a neighbor detects that a node is faulty, *node-level recovery* is initiated. We assume that there is some reconfiguration algorithm that supplies the recovery algorithm with destination node(s) to which processes are to be restored. Since no information is available from the failed node(s), recovery

trees must be constructed based on information available outside the failed node(s). An additional phase is added to the general structure of the algorithms to collect the missing information — namely, the first-level lists (see Procedure *Rback* in Section V). In this phase, each neighbor which has detected the node failure starts up a Recovery Initiator Process ( $RIP_Y$  where  $Y$  is the *node* ID on which  $RIP_Y$  is running). The main tasks performed by this process are: determining which processes were on the failed node(s); determining the first-level list for each failed process; and starting up Recovery Coordinators (phase two) for *each failed process*. Phase two is identical to recovery from communication errors.

In order to determine which processes were on the failed node we require that the immediate neighbors of each node keep track (a list) of all processes running on the node. It should be noted that the disks must have lists of all processes in the system (the union of all the disk server process tables see Figure 3 in the next subsection). Hence, if multiple nodes should fail simultaneously we can determine which processes are still in existence on the system and hence, which processes have failed. After determining which processes have failed,  $RIP_Y$  needs to get information equivalent to first-level process lists, for *each failed process*, before tree construction can begin. This is done by broadcasting RECOVERY messages (containing the failed process list) to all working nodes in the system. Each node returns a “FIRST-LEVEL message” which contains a list of processes on that node who have communicated directly with processes on the failed node. With this information all the required recovery trees can be constructed and recovery can proceed as in the previous subsection.

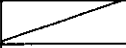


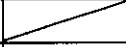
### C. Handling Failures during Checkpoint Sessions

As discussed in Section VII.1, a recovery session always has precedence over a checkpointing session. Specifically, checkpointing handlers stop all checkpointing activity immediately upon receiving a ROLLBACK message, and join the recovery session. Hence, for example, if the checkpointing coordinator receives a ROLLBACK message while waiting for CH\_DONE messages from its children, it will never send a CH\_COMMIT but rather, it will immediately forward the ROLLBACK message to all its children.

On each disk node there is a disk server process that saves and restores checkpoints from the disk. The server process maintains a table with information regarding the status of the checkpoint of each application process whose state is stored on the disk (see Figure 3). Normally, a process using the disk for checkpoint storage has at most one entry, or checkpoint. During a checkpoint session, a second entry is made as the new checkpoint state begins to arrive at the disk node. This entry in the process table is invalid until the last state packet is received.

For each process in the system there is a *version* variable stored on the node where the process is

**Disk Server Process Table**

	<i>PID</i>	<i>Valid</i>	<i>New</i>	<i>Current CCid</i>	<i>Disk Address</i>	<i>Pointer</i>
A)	<i>F</i>	1	0	<i>B</i>		
B)	<i>F</i>	1	0	<i>B</i>		
		0	1	<i>A</i>		
C)	<i>F</i>	1	0	<i>B</i>		
		1	1	<i>A</i>		
D)	<i>F</i>	1	0	<i>A</i>		

**Figure 3:** The process table maintained by the disk-server process running on the disk node.

A) shows an entry for process *F* between checkpoint sessions. Process *B* was *F*'s previous checkpoint coordinator and the disk address field points to the location on disk where *F*'s state is stored. B) shows process *F* in the midst of a checkpoint session which is coordinated by process *A*. In C) *F* has two valid process states on disk and, in D), the newest state has been committed to.

executing[23]. If an error is detected and recovery is necessary, this variable is used to determine which version of the process checkpoint on disk should be used. The version variable has three possible values: *known*, *old*, and *unknown*. During normal operation, the version is always “known”, meaning that there is only one valid checkpoint saved on disk. When the handler begins sending the new checkpoint state of a process to disk, the value of the version variable changes to “old”. When the message queue is sent to disk, the handler changes the version to “unknown” and waits for a CH\_COMMIT message. When the CH\_COMMIT message is later received the handler sends a COMMIT message to the disk node and waits for an acknowledgement. When the acknowledgement is received, the version is changed back to “known”.

The version variable associated with the checkpoint coordinator never changes since the version is always “known”, i.e., there is never more than one *valid* checkpoint state on disk. This means that the entry in the process table associated with the checkpoint coordinator never passes through step C in Figure 3, but moves directly from B to D. Thus the entire interacting set is actually committed to the new checkpoint when the checkpoint coordinator's entry in the disk server's process table changes from B to D. This makes the checkpoint algorithm robust to failures. If any or even all processes in the interacting set fail, recovery coordinators can look at the disk entry for the process being recovered — if there are

two valid states for that process then the coordinator's disk entry is used to determine which state to roll back to. If the coordinator's entry has two states on disk, one valid and one not, then the process rolls back to its older checkpoint, while if there is only one valid entry the process rolls back to its newer checkpoint.

## X. Minimizing the Cost of Checkpointing Using the Virtual Memory System

Asynchronous process-level checkpointing is amenable to several optimizations that can dramatically decrease the amount of data that must be moved during checkpointing as well as the local memory space needed for the volatile checkpoints. The key to these optimizations is to use an enhanced virtual memory system. This system will help the scheme in two ways: 1) it will identify the pages that have been modified since the last checkpoint and thus eliminate the need to checkpoint pages that have not changed, and 2) it will allow the volatile checkpoint to be taken by simply copying page table entries without actually moving data in local memory unless it becomes necessary.

The first optimization requires adding a special "dirty bit" in each entry in the page table. This dirty bit is cleared when a process is checkpointed and is set whenever there is a *write* to the page. When a page is first allocated, it is marked as "dirty." During checkpointing, only dirty pages that are in local memory need to be physically copied to stable storage for checkpointing. Pages that are "clean" according to this special dirty bit have not changed since the last checkpointed and are included in the new checkpoint in stable storage using simple pointer manipulation there (there is no need to move the page from the physical location in stable storage where it was stored for the previous checkpoint).

In standard virtual memory systems, page table entries include control bits that mark the page as *read-only*, *read-write*, *execute-only*, etc. The time to take the volatile checkpoint as well as the space required for it in local memory can be cut dramatically by adding another type of page: *copy-on-write* [3]. Taking a volatile checkpoint involves copying the page table entries of resident dirty pages and marking the corresponding entries in the page table of the process being checkpointed as *copy-on-write*. As long as the process does not actually attempt to write into the page, the page will *not* be copied to the volatile checkpoint area in local memory. Instead, as long as the process is only reading the page, the checkpointing handler, that is copying the page to stable storage, and the process using the page for normal processing will share the page. If the process tries to write to the page, a special page fault will be triggered and the page will be copied to a different location in memory. On the other hand, if the checkpointing session completes before the process attempts to write to the page, the page table entry is restored to its previous value and any writes by the process will *not* trigger a page fault. It should be noted that this very promising optimization can *only* be used with asynchronous checkpointing. Previous

checkpointing techniques block process execution until checkpointing is complete so that "sharing" of pages between the process and the checkpointing handler is meaningless.

In a typical system pages containing code are usually read-only, loads are more frequent than stores, and that a checkpointing session is expected to take significantly less time than the interval between checkpoints. Based in these facts we expect the optimizations described above to significantly reduce both the time to take a volatile checkpoint and the time to commit the checkpoint to stable storage.

## **XI. Conclusions**

We have presented a new distributed error recovery scheme for multicomputer that minimizes disruptions to normal processing due to checkpointing. The scheme is based on coordinated volatile checkpointing of interacting sets of processes followed by copying of the checkpoints to stable storage "in the background" involving minimal interference with normal computation. The scheme is integrated with an efficient error detection mechanism that avoids the need for message acknowledgements, transmission of message sequence numbers, and transmission of check bits with each message. The scheme can recover from multiple node failures, lost messages, and corrupt messages. Multiple checkpointing and recovery sessions may be active in the system simultaneously operating independently if possible and merging correctly when necessary.

We have shown that message logging is not an appropriate error recovery techniques for high-performance multicomputers in which fine-grain parallelism is exploited. On the other hand, the proposed asynchronous process-level checkpointing is amenable to highly-effective optimizations that can be supported with relatively small enhancements to a conventional virtual memory system. With these optimizations, a volatile checkpoint can be taken by only a few dozen instructions without physically copying even a single page in local memory. Hence, the disruption to normal processing will last, at most, tens of microseconds. The amount of data that has to be sent to stable storage is minimized and consists only of pages that have been modified since the last checkpoint and are resident in local memory when the checkpointing session is initiated. Using the proposed techniques, it is possible to implement a highly reliable, general-purpose, large multicomputer system in which the fault tolerance characteristics involve minimal performance overhead and are completely transparent to the user.

## Appendix: Message Types and Their Contents

### *CHECKPOINT and CH\_ACK:*

SenderID, ReceiverID  
Checkpoint Coordinator ID ( $CC_{id}$ )  
Virtual Circuit signature which is denoted as Ssig[receiver] (kept by Sender) or Rsig[sender] (kept by Receiver)  
CH\_ACK also has Child/NotChild flag

### *SUBTREE, CH\_FOUND, CH\_DONE, CH\_COMMIT, and CH\_RESUME:*

SenderID, ReceiverID,  $CC_{id}$

### *RECOVERY:*

SenderNodeID, ReceiverNodeID  
Recovery\_Initiator\_Process ID (RIPid)  
Failed Node ID (i.e. a unique error code)  
List of processes which were running on the failed node

### *FIRST\_LEVEL:*

SenderNodeID, ReceiverNodeID  
Recovery\_Initiator\_Process ID (RIPid)  
for processes  $X = X_1, X_2 \dots X_N$  where N is the number of processes which were running on the failed node:  
ProcessList(X) = ProcessIDs (node,task,process) of processes which sent or received messages from X

### *ROLLBACK and RE\_ACK:*

SenderID, ReceiverID  
Recovery Coordinator ID (RCid)  
VERSION (known, old or unknown)

### *VERSION:*

SenderID, ReceiverID  
VERSION (known, old or unknown)

### *RE\_DONE, RE\_RESUME, and RE\_RES\_ACK:*

SenderID, ReceiverID

### *VERSION\_REQUEST:*

SenderID, DISK\_SERVER ID  
Checkpoint Coordinator ID

### *VERSION\_REQ\_ACK:*

DISK\_SERVER ID, ReceiverID  
Checkpoint Coordinator ID  
VERSION (known, old or unknown)



## References

1. G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).
2. D. Bertsekas and R. Gallager, *Data Networks*, Prentice Hall (1987).
3. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM* 15(3), pp. 135-143 (March 1972).
4. A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).
5. K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems* 3(1), pp. 63-75 (February 1985).
6. W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a Message-Driven Processor," *14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 189-196 (June 1987).
7. W. J. Dally, "Fine-Grain Message Passing Concurrent Computers," *Proceedings of the Third Conference on Hypercube Concurrent Computers*, Pasadena, CA 1, pp. 2-12 (January 1988).
8. S. A. Elkind, "Reliability and Availability Techniques," pp. 63-181 in *The Theory and Practice of Reliable System Design*, ed. D. P. Siewiorek and R. S. Swarz, Digital Press (1982).
9. T. M. Frazier and Y. Tamir, "Application-Transparent Error-Recovery Techniques for Multicomputers," *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Monterey, CA 1, pp. 103-108 (March 1989).
10. W. J. Jager and W. M. Loucks, "The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System," *1987 International Conference on Parallel Processing*, St. Charles, IL, pp. 600-609 (August 1987).
11. D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 14-19 (July 1987).
12. R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering* SE-13(1), pp. 23-31 (January 1987).
13. B. W. Lampson and H. E. Sturgis, "Crash Recovery in a Distributed Storage System," Technical Report, Xerox PARC, Palo Alto, CA (April 1979).
14. P. M. Merlin and B. Randell, "State Restoration in Distributed Systems," *8th Fault-Tolerant Computing Symposium*, Toulouse, France, pp. 129-134 (June 1978).
15. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* 10(2), pp. 123-165 (June 1978).
16. D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, The MIT Press (1987).
17. F. B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Transactions on Computer Systems* 2(2), pp. 145-154 (May 1984).
18. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* 28(1), pp. 22-33 (January 1985).
19. A. P. Sistla and J. L. Welch, "Efficient Distributed Recovery Using Message Logging," *Eighth ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, Canada, pp. 223-238 (August 1989).

20. R. E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems* 3(3), pp. 204-226 (August 1985).
21. R. E. Strom, D. F. Bacon, and S. A. Yemini, "Volatile Logging in n-Fault-Tolerant Distributed Systems," *18th Fault-Tolerant Computing Symposium*, Tokyo, Japan, pp. 44-49 (June 1988).
22. Y. Tamir and C. H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).
23. Y. Tamir and C. H. Séquin, "Error Recovery in Multicomputers Using Global Checkpoints," *13th International Conference on Parallel Processing*, Bellaire, MI, pp. 32-41 (August 1984).
24. Y. Tamir and T. M. Frazier, "Application-Transparent Process-Level Error Recovery for Multicomputers," *Hawaii International Conference on System Sciences-22*, Kailua-Kona, Hawaii, pp. 296-305, Vol I (January 1989).
25. A. S. Tanenbaum, *Computer Networks*, Prentice Hall (1981).
26. C. Whitby-Strevens, "The Transputer," *12th Annual Symposium on Computer Architecture*, Boston, MA, pp. 292-300 (June 1985).