

## APPLICATION-TRANSPARENT PROCESS-LEVEL ERROR RECOVERY FOR MULTICOMPUTERS †

*Yuval Tamir and Tiffany M. Frazier*

Computer Science Department  
University of California  
Los Angeles, California 90024  
U.S.A.

### Abstract

A *multicomputer* system consisting of hundreds of processors interconnected by point-to-point links can achieve high performance for many important applications. We propose a new application-transparent, process-level, distributed error recovery scheme for multicomputers. Checkpointing is initiated by timers at intervals determined by the needs of the application. Checkpointing and recovery involve only as much of the system as is necessary: a set of interacting processes. Processes which are not part of the interacting set do not participate in checkpointing or recovery and continue to do useful work. Several checkpoint and/or recovery sessions may be active simultaneously. The scheme does not require significant overhead during normal operation since it is not necessary to make message transmission atomic, acknowledge each message, or transmit check bits with each packet. We discuss variations of our technique using packet-switching or virtual circuits, and compare our scheme to previously published techniques.

### I. Introduction

*Multicomputer* systems, consisting of hundreds of processors interconnected by high-speed links, are now technologically and economically feasible [12, 20]. Such systems can achieve high performance for many applications at a relatively low cost. 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 [17]. To this end we propose a new application-transparent low-overhead fault tolerance scheme for multicomputers based on *process-level* error recovery. With this technique both the checkpointing and recovery algorithms involve only as much of the system as is necessary: a set of processes that have interacted since their last checkpoint [1]. Processes which are not part of this *interacting set* need not participate in checkpointing/recovery and may continue to do useful work.

Checkpointing is initiated by “timers” associated with each process [1]. The frequency of checkpointing can thus be tuned to the specific needs of a task: a higher frequency of checkpointing results in higher overhead but in less work being lost when recovery is necessary. Thus, different applications running on the same system at the same time can be checkpointed at different frequencies. Checkpointing involves saving a consistent snapshot of the states of an interacting set of processes [2, 17]. The state of each process can be saved either in the memory of a neighboring node or on disks which are connected to a subset of the nodes in the system (henceforth called *disk nodes* [17]).

In the proposed scheme checkpointing and recovery are done at the level of processes with no system-wide central coordination. Many checkpointing and recovery sessions may be active simultaneously. Unrelated sessions do not interfere with each other, while the actions of related ones are properly coordinated.

Performance and storage overhead during normal operation is minimized at the cost of a potentially expensive recovery (rollback). Efficient use of the communication network is ensured by using *signature registers* [17] to avoid the need for message acknowledgements or for sending check bits with each message. Furthermore, there is no “book-keeping” information that must be sent with each message or extensive logs that must be maintained in parallel with normal operation. The scheme poses no restrictions on the behavior of application software and is thus useful for “general purpose” environments where the programmer is unaware of the fault tolerance characteristics of the system.

In this paper we present a complete fault tolerance scheme. We describe the use of the proposed error recovery technique in conjunction with practical low overhead error detection mechanisms. The effects of the message transport mechanism (see Section IV) on error detection and recovery are also discussed. We begin, in Section II, with a brief description of the problem and the requirements from the solution. Several critical basic concepts used in this work are described in Section IV.

† This research is supported by TRW Corporation and the State of California MICRO program.

Section V is a brief description of the error detection techniques we employ. A high-level description of the proposed algorithms is provided in Sections VI, VII, and VIII. A comparison with previous work on error recovery in multicomputers is presented in Section IX.

## II. Fault Tolerance for Multicomputers

Our process-level error recovery scheme is designed to work with multicomputers consisting of hundreds or thousands of VLSI 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 destination. 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.

In order to achieve fault tolerance, the system must include facilities for high-coverage error detection. Once an error is detected, the system must be capable of restoring a valid system state and continuing with normal correct operation. Since programming and debugging programs on a large multicomputer is a difficult task, it is highly desirable for the fault tolerance mechanism used to be transparent to application programmers so as not to further complicate their task.

The design of any fault tolerance mechanism involves a tradeoff of coverage, worst-case duration of interruption of service, hardware overhead, and performance overhead during normal computation. Many of the applications executed on large multicomputers are simulations and numerical computations that are “batch” in nature and do not have strict real-time constraints. For these applications, loss of many minutes of computation when an error occurs may not be important as long as average system performance remains high [17]. For more interactive applications additional performance overhead may be justified in order to prevent the possibility of losing the results of the last few minutes of interactions with the user (or external device). The fault tolerance scheme should thus be “configurable” to meet the requirements of each application while not penalizing all applications equally in order to satisfy the requirements of the most demanding application.

## III. Assumptions

The protocols described in this paper are based on several simplifying assumptions. Many of these assumptions are similar to those discussed in [17] and can be relaxed in similar ways to those discussed there.

We assume a closed system that consists of nodes, links, and disks. Input is stored on disk before operation begins. Output is stored on disk when the job ends.

The nodes are self-checking and are guaranteed to signal an error to their neighbors immediately when they send incorrect output [16]. Any node that generates an error signal is assumed to be permanently faulty and no attempt is made to continue to use it.

Hardware faults either cause a node to generate an error signal or cause an error in transmission.

In order to minimize network traffic, the average number of “hops” from each node to the nearest disk should be small. Hence, disks are connected to several nodes throughout the system. We will assume that a failure in disks themselves or in the disk nodes causes a *crash* (i.e., an unrecoverable error).

Each node has a unique identifier and there is a total ordering of these identifiers. Each process on a node has a unique identifier and there is a total ordering of process identifiers within a node.

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.

## IV. Basic Concepts

The message delivery mechanism has a strong effect on the performance of our checkpointing and recovery schemes. Two mechanisms will be considered: virtual circuits and simple packet switching. With virtual circuits, a logical circuit is set up from the source to the destination by placing appropriate entries in the routing tables of each node [13, 11]. Once the path is set up, there is minimal routing overhead for packets sent through the circuit and FIFO ordering of these messages is maintained. With message/packet switching no path is established in advance between the sender and the receiver. Every packet is routed independently at each hop and packets (messages) may arrive at their destination out of order.

A key idea in this paper is the use of checkpointing and recovery of sets of interacting processes rather than individual processes, individual nodes, or the system as a whole [1]. An *interacting set* of processes is a set of processes that have communicated with each other directly or indirectly since their last checkpoint. The system is composed of a collection of *disjoint* interacting sets of processes. Since there has been no communication between processes in different sets since their last checkpoint, a new checkpoint of a process in one interacting set is consistent with both the old and a new checkpoint of another process which is in some

other interacting set. Hence, different interacting sets may be checkpointed and recovered independently.

The *state* of a process, which gets checkpointed periodically and recovered once an error is detected, 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.

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 and recovery of a particular process is initiated, the process is prevented from further execution and the kernel spawns a special *handler* process that effectively replaces the application process for the duration of the checkpointing or recovery sessions. The handler can manipulate the state of the process and can, as its final action, cause the process to become “executable” again. 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.

## V. 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[16].

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 [19]. 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 [17], 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 [4].

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.

When a node fails, recovery involves rolling back the union of the interacting sets of all the processes that were running on that node and requires eventual roll back of all interacting sets of processes which had any messages in transit on that node at the time of its failure. When an error is detected by a mismatch of the LFSRs on two ends of a physical link and *packet switching* is used, the entire system must be rolled back since there is no way to determine which processes were affected by corrupt messages. If virtual circuits are used, the error is detected by a mismatch of signatures on the two ends of a virtual circuit and only the interacting set which includes the two processes connected by that virtual circuit need to be rolled back.

## VI. Coordinating Interacting Sets as Trees

The proposed error recovery scheme is based on checkpointing consistent states of interacting sets of processes. 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 the processes that were

members of those interacting sets are rolled back to that last checkpoint.

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, *dynamic communication information* which is the list of processes with whom there has been *direct* communication since the last checkpointing session [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.

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. To this end, *CHECKPOINT* and *ROLLBACK* messages include the identifier of the coordinator, which consists of a node and process 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. If a process receives one *CHECKPOINT* and one *ROLLBACK* message, the *ROLLBACK* message “wins” regardless of the coordinator ID. The process propagates the winning *CHECKPOINT* or *ROLLBACK* messages to all its first-level processes, even if it has previously propagated the losing *CHECKPOINT* or *ROLLBACK* messages. Eventually, the step-by-step propagation of the winning session “flushes out” all remnants of the losing session and a consistent communication tree for checkpointing or rollback is established.

As discussed in the next section, during a checkpointing session messages in transit between members of the interacting set are “flushed” to their final destinations and are checkpointed together with these destination processes. Similarly, during a recovery session, messages in transit between members of the interacting set must be discarded from the system together with the processes that are replaced by a previously saved checkpoint. Since the formation of the communication tree is not instantaneous, it is possible for new messages to be sent to or from members of the interacting set during this process. During the formation of the communication tree, a process (X), which is part of the interacting set but not yet part of the communication tree, may initiate communication for the first time with another process (Y), which is already part of the communication tree. The checkpoint generated by this session must include, as part of the state of Y, any messages sent from X to Y before X was incorporated into the communication tree. This implies that the state of Y to be checkpointed is not defined until the communication tree is complete (until the root receives acknowledgements from all its children). The other potential problem occurs if a process (X) which is *not*

part of the interacting set initiates communication with a process (Y) which is already part of the communication tree. In this case any messages sent must *not* be delivered to Y and are held until the checkpointing session is complete.

## VII. Checkpointing

A checkpointing session for a task (set of processes) is usually initiated by the triggering of a “timer” associated with one of the processes. When this timer goes off, a checkpoint coordinator ( $CC_X$ , where X is the process who is initiating the checkpoint) starts up.  $CC_X$  attempts to create a communication tree (*checkpoint tree*), of which it is the root, and coordinate a checkpointing session. The main tasks of the checkpointing algorithm are to: identify the interacting set; make sure the process states are complete by flushing any messages in transit to their destination processes in the interacting set; detect any errors that may exist in any of the process states and, if any, initiate a recovery session; otherwise, save the process states, and resume normal processing. As discussed above, several checkpointing sessions for the same interacting set may be initiated “simultaneously” but they will be appropriately merged into a single checkpoint tree.

Immediately after process X initiates a checkpointing session, its state is changed to “checkpointing” and it does not get any further opportunity to execute until normal operation is resumed. The checkpointing coordinator,  $CC_X$ , takes the place of X in receiving (application-level) messages destined for X. Messages for X are accumulated (by  $CC_X$ ) in a message queue which is checkpointed with X at a later stage of the checkpointing session.  $CC_X$  also begins sending X’s state to the disk node in charge of maintaining X’s checkpoint. As discussed in the previous section,  $CC_X$  initiates the construction of the checkpoint tree by sending CHECKPOINT messages to all of process X’s first-level processes. If communication in the system is based on virtual circuits (Section IV), the CHECKPOINT messages are sent down virtual circuits, if they still exist. Since all communication along a virtual circuit is FIFO, any messages in transit will be flushed to their destinations. The “source” signature of the circuit, which is sent as part of the CHECKPOINT message, is then compared to the “destination” signature. If there is a mismatch a recovery handler will be started and the checkpoint handler will terminate. For circuits that have been “torn down,” at which time the signature comparison would have been made, CHECKPOINT messages are simple routed “hop-by-hop” and carry no signature.

As discussed earlier, the receipt of a

CHECKPOINT message by process Y causes an operating system to spawn a handler process that will take care of all checkpointing duties for that process ( $CH_Y$  - checkpoint handler for process Y). First,  $CH_Y$  flags Y as “checkpointing” and sends CHECKPOINT messages to all the processes with which Y has communicated except for Y’s parent, and so on until the tree is complete, i.e., until the leaf nodes are reached.  $CH_Y$  then begins to send Y’s state to the appropriate disk node and takes over receiving messages destined for Y and accumulating them in the message queue that may be included as part of Y’s checkpoint state later in the session. After sending CHECKPOINT messages, all the handlers (CHs and CC) wait for CH\_ACK (checkpoint acknowledgement) messages from the processes to whom CHECKPOINT messages were sent.

Upon receiving a CHECKPOINT message, processes which are already part of the tree respond with a CH\_ACK(NOT\_CHILD) message to the sender while other processes respond with a CH\_ACK(CHILD) message. Once a process has received CH\_ACK messages for all the CHECKPOINT messages it sent, it sends a CH\_ACK(CHILD) message to the sender of the first CHECKPOINT message it received (its parent).

As discussed in the previous section, the checkpoint tree is complete only after the coordinator has received CH\_ACK messages from all its children. It is only at this point that the message queues being accumulated by the handlers are guaranteed to contain all the messages which were in transit between processes in the interacting set and all the signature cross-checks are complete. Once the coordinator  $CC_X$  receives CH\_ACK messages from all its children, it begins sending X’s message queue to the disk node where it is included with X’s saved state.  $CC_X$  also sends CH\_COMMIT messages to all its children, thus informing them that the tree is complete and that message queues may now be sent to disk.  $CC_X$  then waits for an acknowledgement from disk, signifying the correct transfer of process X’s state and message queue, and for CH\_DONE messages from all its children.

When  $CH_Y$  receives the CH\_COMMIT message, it forwards the CH\_COMMIT to its children, sends Y’s message queue to disk, and waits for the acknowledgement from the disk node (for the message queues and for Y’s state) and for CH\_DONE messages from all its children.  $CH_Y$  then sends a CH\_DONE message to its parent and waits for a CH\_RESUME message. When the  $CC_X$  receives CH\_DONE messages from all its children, it sends a resume message to the disk node where its state is being stored, causing the disk server process (see next subsection) to commit to the

new state. Once  $CC_X$  receives an acknowledgement from the disk, it sends CH\_RESUME messages to all its children. Each of the checkpoint handlers in turn commits the new states and forwards the CH\_RESUME message to its children.

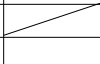


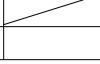
Normal operation cannot be resumed while the CH\_RESUME messages are still being propagated down the tree. If processes are allowed to resume normal operation in the middle of the CH\_RESUME “phase,” then it is possible for part of the checkpointing tree to be resuming normal operation and be committed to the new checkpoint while another part of the tree is still not committed to the new checkpoint (see subsection VIII.B). If a node containing two or more processes that are members of the interacting set fails at this time, the tree may be partitioned into two or more disjoint subtrees: a subtree in normal operation committed to the “new” checkpoint and a subtree that may decide to roll back to the “old” checkpoint. Thus, these subtrees may end up being rolled back to inconsistent states. Hence, normal operation does not resume until the new state has been committed to by *all* members of the checkpoint tree. To this end, CH\_RES\_ACK messages are sent back up the tree by each leaf process after its *resume* has been acknowledged by the appropriate disk node. At this point, the checkpoint handler moves its process into the “ready-to-run” queue and terminates, thus resuming normal operation. In this way, the CH\_RES\_ACK messages proceed up the tree to the coordinator, which is the last handler to terminate.

### A. Disk Nodes

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 1). 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 executing [17]. 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. Just before a handler begins

**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 1:** 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.

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\_RESUME message. When the CH\_RESUME message is later received the handler sends a RESUME 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 process 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 1, 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. As discussed in the next section, this is crucial for recovery from node failure which occurs during a checkpointing session.

### B. Checkpointing with Packet-Switching

In a packet-switching environment messages are routed on a hop-by-hop basis over virtually any path in the system. Flushing of messages in transit during checkpointing is more complex since the entire system must be flushed. This requires temporarily stopping all

the processes in the system from generating new messages while existing messages in transit are flushed to their final destinations and checks are performed on *every* communication link. As described by Tamir and Gafni [18], this is a classic distributed termination problem and the solution we use here is the same one used in [18], which is derived from [14, 3].

### VIII. Recovery

Recovery sessions are initiated when an error is detected. There are two basic types of recovery sessions: recovery of a single interacting set, and *node-level* recovery, which may involve multiple interacting sets. The type of recovery required depends on the nature of the error detected by the system. When the two signatures on the ends of a virtual circuit do not match, a single interacting set needs to be rolled back - that set which contains the two processes on either end of the virtual circuit. Node-level recovery is required when an error in a node's outputs is detected by its neighbors [16]. In this case, recovery may involve more than one interacting set since we assume that no information is available from the node itself and we must also worry about any messages which were being routed through the node when it failed.

#### A. Recovering From Communication Errors

The algorithm for recovery from an error in communication is very similar to the checkpointing algorithm, where recovery coordinators and handlers replace their checkpointing counterparts. When an error in communication is detected (by a mismatch in signatures), the kernel on the node where the error is detected initiates a rollback by spawning a *recovery coordinator* process (handler) that "replaces" the application process associated with the detected error. A "recovery tree" is created by propagating ROLLBACK messages which are acknowledged by RE\_ACK(CHILD or NOT\_CHILD) messages in exactly the same way 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. 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 node 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.

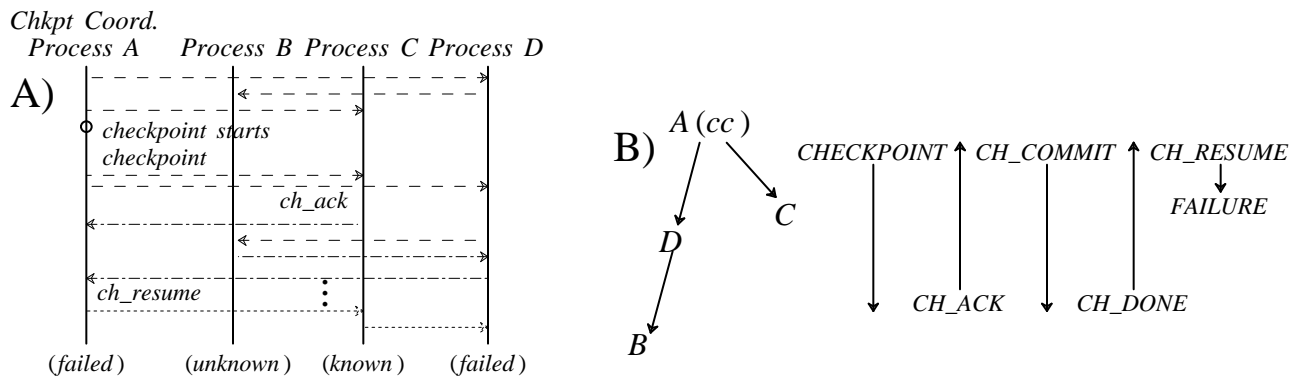
#### B. Node-Level Recovery

Node-level recovery is initiated when a node indicates to its neighbors that it is faulty (as discussed in sections III and V, we assume the nodes are self-checking). Under these conditions it is assumed that the entire node has failed and all information that was stored in it is lost. This node resets itself [16] and, if it is usable, processes may be restored back onto it, otherwise they need to be restored onto other nodes in the system. We assume that there is some reconfiguration algorithm that supplies the recovery algorithm with destination node(s) to which the processes are to be restored.

Since no information is available from the failed node, the obvious "roots" of the relevant recovery trees do not exist. Recovery trees must be constructed based on information available outside the failed node. An additional phase is added to the general structure of the algorithms to collect the missing information. 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; determining the first-level list for each failed process; and starting up Recovery Coordinators (phase two) *for each failed process*. Phase two is almost identical to recovery from communication errors. Due to lack of space, the entire node-failure algorithm will not be presented here but critical problems will be discussed.

It should be noted that when a node fails, messages in transit that happened to be in the node at the time of failure will be lost. While it is possible to detect the fact that messages have been lost as part of the recovery process, there would usually be no real need to do so. The loss of messages due to node failure is equivalent to any other error in communication and is already dealt with by the checkpointing and recovery algorithms.

**Constructing First-Level Lists.** The immediate neighbors of each node keep track (a list) of all processes currently running on the node. Whenever a process is initiated during normal operation, a fail-safe message relaying this fact is sent to the neighboring nodes. When a process terminates, it initiates a checkpoint session at which time the process's ID is removed from the neighboring nodes' lists. After retrieving this list,  $RIP_Y$  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



**Figure 2:** **A)** An example of an interacting set. Horizontal arrows indicate communication between two processes while time flows from top to bottom. A checkpointing session is initiated by process A and a failure of the node on which process A and D are running occurs during the checkpoint session. **B)** The corresponding checkpointing communication tree and the flow of messages during checkpointing until the error is detected.

messages (containing the failed process list) to all 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.

**Recovery During a Checkpoint Session.** In order to be able to recover from errors that occur, or are detected, during a checkpoint session, it must be possible to determine which process state to roll back to: the state being saved or the state saved previously. As described earlier, with each process in the system there is an associated *version* variable which is stored locally on the node on which the process is executing [17]. During recovery sessions this variable is examined in order to determine which checkpoint to use.

The problem with a recovery session being initiated in the middle of a checkpointing session is complicated by the fact that parts of the interacting set can be lost when a node fails (see Section VII). This problem is illustrated in Figure 2 where processes A and D were running on the same node when that node failed. Since all communication information (first-level lists) are lost, there is no way to know that processes A and D had communicated with each other. Hence,  $RC_A$  and  $RC_D$  will each have their own recovery tree when, in fact, they belong to the same interacting set. Depending on the states of the processes’ version variables within these trees, the recovery coordinators may independently decide to roll back to opposite versions - leading to an incorrect system state.

During recovery, if any of the processes in a recovery tree have “old” version variables, then the entire tree rolls back to the oldest (or only) version. If any of the processes in the tree have “known” version variables, then the entire tree rolls back to the newest version. The checkpointing process described in

Section VII guarantees that within an interacting set it is not possible for one process’s version variable to be “old” while another process’s is in the “known” state. Hence, there is a problem only if ALL processes in the recovery tree are in the “unknown” state. In this case the Recovery Coordinator accesses the disk information associated with the checkpoint coordinator, see Figure 1, and determines which version it’s recovery tree must roll back to. This will always work no matter how much of the interacting set is on the failed node. Once the disk node commits to the newest process state for the checkpoint coordinator, the entire interacting set has committed to the newest state.

**C. Recovery in a Packet Switching Environment**

If communication is based on packet switching rather than virtual circuits, rollback may be slightly more complex. Specifically, before rollback can occur *all* messages in transit in the system must be flushed to their final destinations. Those messages whose final destinations are processes that are being rolled back are then discarded. If each process keeps a separate signature of its communication with every other process, errors in communication can be detected by comparing corresponding signatures.

Without maintaining separate signatures of communication with every other process, the entire system must be rolled back whenever an error is detected. In such an environment LFSRs can be used at the link level to detect errors in communication between neighboring nodes. As part of the process of flushing messages in transit, the signatures in these LFSRs are compared. If a mismatch of signatures is found, there is no way to pinpoint when the error occurred or which processes were affected by it. If it is impossible to determine which processes were affected by the error, the entire system must be rolled back. Similarly, if a node fails, messages in transit may be lost and it is not



possible to determine the source or destination of these messages. Here again, the only solution (assuming our environment where messages are not acknowledged by the destination process) is to roll back the entire system.

### IX. Previous Work

Many fault tolerance techniques for multicomputers have been presented in the literature. These techniques can be evaluated in terms of the time and storage overhead during normal operation, the time lost when recovery occurs, and the degree to which the scheme restricts the actions of the application or requires the application to contain special features (such as recovery blocks [10]) for fault tolerance.

Some existing schemes require maintaining multiple checkpoints of each process [21] which may, in the worst case, lead to the rolling back of processes to their initial state due to *domino effect* [10]. Other techniques require that the system be structured out of atomic actions [9] or are based on application software using *recovery blocks* for error detection [8, 21]. Systems where fault tolerance is based on transactions [9] require *serializability* which can significantly degrade the level of concurrency achievable in a system [15]. Existing schemes often restrict the patterns of communication between processes [5] and require significant overhead during normal operation for maintaining the information necessary for recovery. The problems of errors in communication are often ignored by explicitly stated assumptions [8]. In other cases message sending and receiving must be *atomic* [1], so that communicating processes always appear as either having completed a communication or as having not yet initiated it. This requires overhead during normal operation due to the use of a two-phase commit protocol for inter-node messages.

Barigazzi and Strigini 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 [1]. The critical feature of this procedure is that all interacting processes are checkpointed together, so that their checkpointed states can be guaranteed to be consistent with each other. Therefore, the *domino effect* can not 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 [17] 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 checkpoints 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 (on the order of thirty minutes). Hence, the system can only be used for batch applications, such as large numerical computations, where the possibility of losing thirty minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent [17]). In this paper we extend the scheme in [17] to perform checkpointing and recovery of sets of interacting processes rather than of the entire system. This extension results in a scheme that is useful for a system running a variety of tasks, including batch tasks and tasks which require more frequent checkpointing.

A new technique for distributed error recovery called "sender-based message logging" has been proposed recently [6]. This scheme is based on recording all messages a process sends, so that they can be "played back" to a failed, rolled-back process in order to bring that process back to a state consistent with that of the rest of the processes in the system. This technique results in a very fast recovery since only the failed process needs to repeat the computation since its last checkpoint.

A major disadvantage of the original version of sender-based message logging is that it limits concurrency by prohibiting a process from sending any messages until it has been notified that the senders of all previous messages to it have logged the messages they sent [6]. An alternative *optimistic* protocol for sender-based logging eliminates the concurrency loss at the expense of requiring the broadcast, before each checkpoint, of a message whose size is proportional to the number of processes with which the checkpointing process has communicated. The double

acknowledgement of each message in either protocol significantly increases message traffic. In addition, since all messages sent are logged by the sender, there is a performance and space overhead incurred during normal operation associated with the logging process.

Since the valid recovery of a process is dependent on the message logs of the processes within its interacting set, sender-based logging is not easily extendible to handle the failure of more than one process at a time. If, as is the case with our technique, the only information needed for recovery is stored during checkpointing, it is possible to save this information in multiple locations (e.g. "mirrored" disk drives [17, 7]), thus supporting recovery from multiple node failures. Since message-logging requires the saving of messages during every send, saving messages on non-local and/or slower devices would degrade performance significantly.

### X. Summary and Conclusions

A general-purpose recovery scheme, integrating error detection facilities with checkpoint and recovery algorithms, has been presented. This scheme takes into account the difficulties which arise when attempting to ensure the correct transmission of messages in a multicomputer system. The proposed technique handles total node failures where several processes and messages in transit may be lost simultaneously. The algorithms presented are most effective in conjunction with virtual circuits, but can also be used in other environments, such as packet-switching, with a significant degradation in the speed of checkpointing and recovery.

By not performing checkpoints at a level lower than an interacting set of processes, our scheme incurs little overhead during normal operation, is free of domino effect, and is completely application transparent. Since checkpointing and recovery are performed on interacting sets, and do not involve the rest of the system, several checkpointing and recovery sessions can be active at the same time, while other parts of the system are performing their normal operation, without compromising correctness. Using the proposed techniques, it is possible to implement a highly reliable, general-purpose, large multicomputer system in which the fault tolerance characteristics are completely transparent to the user.

### 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. 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).
3. E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters* **11**(1), pp. 1-4 (August 1980).
4. 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).
5. S. H. Hosseini, J. G. Kuhl, and S. M. Reddy, "An Integrated Approach to Error Recovery in Distributed Computing Systems," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 56-63 (June 1983).
6. David B. Johnson and Willy Zwaenepoel, "Sender-Based Message Logging," *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 14-19 (July 1987).
7. J. A. Katzman, "The Tandem 16: A Fault-Tolerant Computing System," pp. 470-480 in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill (1982).
8. K. H. Kim, J. H. You, and A. Abouelnaga, "A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes," *16th Fault-Tolerant Computing Symposium*, Vienna, Austria, pp. 130-135 (July 1986).
9. E. Nett, R. Kroger, and J. Kaiser, "Implementing a General Error Recovery Mechanism in a Distributed Operating System," *16th Fault-Tolerant Computing Symposium*, Vienna, Austria, pp. 124-129 (July 1986).
10. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2), pp. 123-165 (June 1978).
11. D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, The MIT Press (1987).
12. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1), pp. 22-33 (January 1985).
13. C. H. Séquin, A. M. Despain, and D. A. Patterson, "Communication in X-Tree, a Modular Multiprocessor System," *1978 Annual Conference of the ACM*, Washington, D.C., pp. 194-203 (December 1978).
14. N. Shavit and N. Francez, "A New Approach to Selection of Locally Indicative Stability," *13th ICALP (Lecture Notes in Computer Science 226)*, pp. 344-358, Springer-Verlag (1986).
15. R. E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems* **3**(3), pp. 204-226 (August 1985).
16. Yuval Tamir and Carlo H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).
17. Yuval Tamir and Carlo H. Séquin, "Error Recovery in Multicomputers Using Global Checkpoints," *13th International Conference on Parallel Processing*, Bellaire, MI, pp. 32-41 (August 1984).
18. Yuval Tamir and Eli Gafni, "A Software-Based Hardware Fault Tolerance Scheme for Multicomputers," *1987 International Conference on Parallel Processing*, St. Charles, IL, pp. 117-120 (August 1987).
19. A. S. Tanenbaum, *Computer Networks*, Prentice Hall (1981).
20. C. Whitby-Strevens, "The Transputer," *12th Annual Symposium on Computer Architecture*, Boston, MA, pp. 292-300 (June 1985).
21. W. G. Wood, "A Decentralized Recovery Control Protocol," *11th Fault-Tolerant Computing Symposium*, Portland, Maine, pp. 159-164 (June 1981).