APPLICATION-TRANSPARENT PROCESS-LEVEL ERROR RECOVERY FOR MULTICOMPUTERS

Yuval Tamir
Tiffany Gaber

January 1988
CSD-880005

Application-Transparent Process-Level Error Recovery for Multicomputers

Yuval Tamir and Tiffany Gaber

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

A *multicomputer* system consisting of hundreds of processors interconnected by high-speed dedicated 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: an interacting set of one or more processes. Processes which are not part of the interacting set need not participate in checkpointing/recovery and may 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.

Keywords: rollback, distributed error recovery, multicomputers.

January 1988

1. Introduction

Due to advances in technology, it is now feasible to implement *multicomputer* systems consisting of hundreds of processors interconnected by high-speed dedicated links.[13, 18] Such systems can achieve high performance for a large class of important applications at a relatively low cost. The reliability requirements of large multicomputers can only be met using fault tolerance techniques. To this end we propose a new application-transparent low-overhead fault tolerance scheme for multicomputers based on *process-level* 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 (communicated) 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.

During normal operation, checkpointing is initiated by "timers" associated with each process on each node.[1] The frequency of checkpointing can thus be tuned to the specific needs of a task: a higher frequency of checkpointing results in a higher overhead but in less work being lost when recovery is necessary. Different applications running on the same system at the same time can be checkpointed at different frequencies. The checkpointing process involves saving the state of the entire interacting set of processes. 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*[16]). Depending on the cause of the error, our recovery algorithm restores a single interacting set, several interacting sets, or, in the extreme case, system-wide rollback is performed. Several checkpoint and/or recovery sessions may be active at the same time.

A key feature of the proposed scheme is that it minimizes the performance and storage overhead during normal operation at the expense of potentially (worst case) expensive recovery (rollback). Using a previously published technique[16] there is no need for message acknowledgement or for sending check bits with each message, thus ensuring efficient use of the communication network. 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 discuss how our error recovery technique is used in conjunction with practical low overhead error detection mechanisms. The effects of the message delivery mechanism (see Section 4) are also described. We begin with a discussion of previously published techniques and compare them to the scheme proposed in this paper, thus providing the motivation for our scheme. A few basic definitions and assumptions are presented in Section 3. Section 4 explains some basic concepts used in this work. Section 5 is a brief description of the error

detection techniques we employ. Section 6 describes the checkpointing and recovery protocols. Variations of the protocols for systems with virtual circuits and simple packet switching (see Section 4) are considered. Special attention is paid to the problem of handling simultaneous interacting checkpointing sessions, errors detected in the middle of checkpointing, and handling of processes which attempt to "join" the interacting set in the middle of checkpointing.

2. Previous Work

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

Several previously proposed recovery schemes are suitable for systems where interprocessor communication is over a common bus or Ethernet.[2, 11] These techniques rely on the ability to monitor and record all interprocessor communication as it occurs and are thus not suitable for multicomputers.

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[12]) for fault tolerance.

Some existing schemes require maintaining multiple checkpoints of each process[19] which may, in the worst case, lead to the rolling back of processes to their initial state due to *domino effect.*[12] Other techniques require that the system be structured out of atomic actions[4] or are based on application software using *recovery blocks* for error detection.[10, 19] Existing schemes often restrict the patterns of communication between processes[7] 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.[10] 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* that may require backing up to successively older states[12] cannot occur. As a result, 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 will not have a detrimental effect on system performance.

The idea of checkpointing and recovering interacting sets of processes is extended in [16] 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 loosing thirty minutes of computation during recovery is an acceptable price for the resulting low overhead (a few percent[16]). In this paper we extend the scheme in [16] 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 whose requirements prohibit the loss of many minutes and therefore require more frequent checkpointing.

A new technique for distributed error recovery called "sender-based message logging" has been proposed recently.[8] 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 to send any messages until it has been notified that the senders of all previous messages to it have logged the messages they sent.[8] 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 whom 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 overhead incurred during normal operation for the logging process and there is space overhead incurred due to the use of some of the node's memory to log messages.

Since the valid recovery of a process is dependent on the correctness of the message logs of the processes within it's interacing set, it seems that sender-based logging is not easily extendable to handle more that one "simultaneous" error in the system. 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[16,9]), 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.

3. Assumptions

The protocols described in this paper are based on several simplifying assumptions. Many of these assumptions are similar to those discussed in [16] 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.[15] 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.

Since the disks are extensively used for paging, checkpointing, and I/O, 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.

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.

4. Basic Concepts

As we will see later in this paper, 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.[6] 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. With message/packet switching no path is established in advance between the sender and the receiver. Instead, when a message is sent, it is stored in each intermediate node and is forwarded, one hop at a time, until it reaches its destination. Every packet is routed independently at each hop and messages may arrive at their destination out of order.

One of the key ideas used in this paper is 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. By definition, at any point in time, different interacting sets in the system are disjoint. Different interacting sets may be checkpointed independently since, by definition, there has been no communication between processes in different sets since their last checkpoint so that a new checkpoint of one interacting set is consistent with both the old checkpoint of the other interacting set and a possible new checkpoint of that second interacting set.

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.

A "recovery set", is the smallest set of processes that must roll back when a error is detected. This is simply the set of all the processes that have interacted with the failed process since its last checkpoint.

We refer to our checkpointing and recovery protocols as being "process-level" since we decide to checkpoint or recover complete processes as indivisible units rather than parts of processes or complete nodes. In a "node-level" scheme the entire state of a node would be treated as an indivisible unit that is checkpointed and recovered as a whole. Node-level checkpointing and recovery would involve much larger "interacting sets" than process-level protocols since the interacting set of a node includes all the

nodes which have communicated with it directly or indirectly since their last checkpoint. Much of the flexibility of tuning the checkpointing frequency to the needs of the application would be lost in a node-level scheme.

5. Error Detection

As previously discussed, errors in the system may be a result of node failures or failures in the communication links. Node failures are detected since the nodes are self-checking and produce an error indication whenever their outputs are incorrect. 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 rolling back all interacting sets of processes who had any messages in transit on that node at the time of its failure.

In most systems, errors in message transmission are detected by including redundant *check bits* with each message. The receiver uses these check bits to determine whether the contents of a message has been corrupted. The disadvantage of this technique is that the transmission of the check bits "wastes" 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 [16], 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.[5]

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. Thus, the communication between processes in the interacting set can be checked without checking all the communication links in the system, but instead, 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 that are used only for transmitting information between kernels which include a sufficient number of redundant bits so that any likely error in transmission will be detected. These special packets are called *fail-safe* packets since they are either error-free or the error is easily detectable by the receiving node.

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 nodes 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.

6. Process-Level Checkpointing and Recovery

In order to perform checkpointing at a process level and avoid the domino effect we must be able to identify an interacting set of processes. Determining this set requires the dynamic storage of communication information with each process. Barigazzi and Strigini[1] proposed the use of boolean communication vectors (CVs) for this purpose. Communication vectors are boolean truth tables where bit Y is set in process X's vector if process X has communicated with process Y since X's last checkpoint. Thus, at the time of a checkpoint, we know all the processes a particular process has communicated with simply by examining it's communication vector. To find interacting sets of processes a communication graph can be formed from all the CVs. This graph is valid only at a particular time t if it is constructed from CVs which have been updated to reflect all communications up to time t. Each node in this graph represents a process and an arc between two nodes X and Y means that processes X and Y have communicated with each other at least once. When the graph is completed it will consist of one or more non-overlapping (sub)graphs - corresponding to the different interacting sets currently present in the system. Since there may be any number of interacting sets present in the system, there may be several checkpoint and/or recovery sessions active at the same time.

6.1. Checkpointing with Virtual Circuits

Checkpoint sessions may be initiated by any process when it's "timer" goes off. This causes a checkpoint coordinator (CC_X where X is the process id of the process initiating the checkpoint) to start up. CC_X creates and coordinates the disjoint portion of the communication graph within which process X resides. The checkpoint coordinator can be implemented as an operating system routine which takes over

checkpointing duties for the initiating process. Since performing a checkpoint on an interacting set ensures that the processes in the set are consistent with each other (by definition) the main requirements of the checkpointing algorithm are to:

* identify the interacting set,
* make sure the process states are complete by flushing any messages in transit to 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.

CC_X initiates construction of the graph, a tree where the process X is the root, by stopping the normal processing of X and sending CHECKPOINT messages to all processes Y with whom X has communicated since X's last checkpoint. These messages are fail-safe but are sent down virtual circuits, if they still exist, in order to flush any messages which may be in transit. For circuits that have been ''torn-down'' since the last checkpoint session the CHECKPOINT messages are routed ''hop-by-hop''. The receiving of a CHECKPOINT message by process Y causes an operating system routine to wake up and handle all checkpointing duties for that process (CH_Y - checkpoint handler for process Y). First CH_Y stops Y's normal processing and sends CHECKPOINT messages to all the processes with whom Y has communicated (EXCEPT for the process from whom the CHECKPOINT message was received - keeping with the structure of a tree), and so on until the graph is complete - i.e. we have reached leaf nodes. After sending CHECKPOINT messages all the CHs and the CC wait for CH_ACK (checkpoint acknowledgement) messages from the processes to whom CHECKPOINT messages were sent.

Leaf nodes (or processes) are reached when a process has communicated only with the sender of the CHECKPOINT message or has communicated only with processes who are already part of the tree. In the latter case the leaf node receives CH_ACK(NOT_CHILD) messages from all such processes. Processes who are *not* already part of the tree send a CH_ACK(CHILD) message back to their parent. Once the leaf process has received CH_ACK messages for all CHECKPOINT messages sent, if any, it sends a CH_ACK(CHILD) message to the sender of the original CHECKPOINT message it received (the leaf processes' parent). At this point the leaf process knows that all virtual circuits that either begin or terminate with itself have been flushed - thus it's state is complete and may be sent to disk. This follows for all nodes/processes in the tree. Once the last state packet is sent to disk the process waits for a SAVED message from the disk node, then sends a CH_DONE message to it's parent (*after* CH_DONE messages have been received from all it's children) and lastly waits for a CH_RESUME message from

the CC. (Note - this is actually a slight simplification that will be discussed in next section). The CC_X sends CH_RESUME messages to its children once it has received CH_DONE messages from them all and has received a SAVED message from it's disk node. These CH_RESUME messages immediately propagate down to the leaf nodes. When the disk node receives a CH_RESUME message, it commits to the new backups and erases the old ones.

In summary, the main activities of the algorithm are sending messages in waves up and down a tree (interacting set) of processes. CHECKPOINT messages have three purposes: setting up the tree, performing one direction of virtual circuit flushing, and also performing one direction of the message correctness checks (signature comparison). CH_ACKS travel from the leaf nodes to the root (CC_X) and result in the flushing of the rest of the virtual circuits not flushed by CHECKPOINT messages and hence, the rest of the signature comparisons are also completed. By the time all CH_ACKS have been sent and received, we know that *all* process states are complete and consistent with each other. The final correctness check of the process states is performed automatically as the states are sent to disk. The disk node acknowledges the receipt of a complete process state by sending a SAVED message back to that process' checkpoint handler. The checkpoint session is concluded by a wave of CH_DONE messages up the tree which results in the CC starting a wave of CH_RESUME messages down the tree at which time all processes return to normal operation.

6.2. 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. This complicates the flushing of messages in transit during checkpointing, since the entire system must be flushed. We cannot solve this problem by allowing the system to continue to route messages as a checkpoint takes place, since the point at which all messages have reached their destinations cannot be accurately determined. The solution is to stop sending any new messages while flushing all messages existing in the system to their final destinations. As described in detail by Tamir and Gafni,[17] this is a classic distributed termination problem and the solution we use here is the same one used in [17], which is derived from [14, 3].

In summary, no *new* normal messages may be sent while messages are being flushed to their destinations. Any other checkpoint sessions which may be active at this time may continue working, however. Since it is also required that no normal messages be sent during the LFSR check, we can combine those two portions of the algorithm so that only a small interval of time, relative to the time it takes to send a process state to disk, requires that new messages cease being transmitted.

Since message flushing and correctness checks need to be performed on a system level while other aspects of checkpointing are done on a process level it is possible for more than one (identical) correctness check to be taking place at the same time. While this results in some redundant checking, the use of packet-switching already requires us to monitor the checkpoint frequency in the system such that system performance is not degraded due to lack of sufficient time to transmit normal messages. Thus, checkpoints will be farther apart, on the average, thereby reducing the probability of overlapping system correctness checks.

6.3. Recovery with Virtual Circuits

Recovery sessions are initiated when an error is detected. Two types of recovery sessions are required depending on the error detected. For example, process-level recovery is initiated when the two signatures on the ends of a virtual circuit do not match. Node-level recovery is required when a node's outputs are detected by a neighbor as being erroneous. Node-level recovery is very similar to process-level recovery except that no information is available from the node itself (i.e. communication vectors or even a list of processes running on the node) and we must worry about any messages in transit that were on the node when it failed.

6.3.1. Process-Level Recovery

Process-level recovery is initiated when an error in communication is detected. The algorithm to perform process-level recovery is almost identical to the checkpointing algorithm, where recovery coordinators and handlers replace their checkpointing counterparts. The "recovery tree" is created by sending 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. No signature comparisons are made, however, and all messages which are flushed to their destinations are discarded. RH_Y may request process Y's state from the appropriate disk node after all expected RE_ACK messages have arrived and then, after having received RE_DONE messages from all Y's children, send a RE_DONE message to Y's parent. The recovery coordinator sends RE_RESUME messages to it's children once it has received RE_DONE messages from each one of them. Upon receipt of an RE_RESUME RH_Y terminates and process Y resumes normal processing.

6.3.2. Node-Level Recovery

Node-level recovery is initiated when the source of an error can not be determined to be within a single interacting set of processes - i.e. a node's outputs are erroneous. We can make no assumption about these outputs and must therefore assume that the entire node has failed. This node resets itself[15] 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 this decision has been made for us by some known reconfiguration routine such that the recovery algorithm will be supplied with destination node(s) to which the processes are to be restored.

Since no information is available from the failed node, multiple recovery trees must be constructed without root processes. Hence an "independent" Recovery Coordinator (RC_{id}), which has a unique I.D. associated with it, is used. This RC performs the same duties as a process-level RC but does this for more than one interacting set of processes. We assume that all nodes keep lists of processes running on their neighboring nodes, therefore if one or more neighbors detect an error on a node they will all start up an independent RC - which then has immediate access to the list of processes which were running on the failed node. (If more than one neighbor starts an RC, these recovery sets will later be merged into one (see Section 6.5.3)).

After receiving this list, the RC needs to get information equivalent to communication vectors before tree construction can begin. This is done by having the RC broadcast RECOVERY messages (containing the failed process list) to all nodes. These nodes then return FIRST-LEVEL messages which contain a list of processes on that node who have communicated directly with a failed process. With this information the RC can construct, in parallel, all the required recovery trees. From here on recovery proceeds just as in process-level recovery, except that every message must have associated with it the RC_{id} and the process ID of the failed process for which a particular recovery tree is being defined.

6.3.2.1. Lost Messages

When a node fails, messages (packets) in transit that happened to be in the node at the time of failure will be lost. In order to determine whether any packets in transit were lost when a node fails a special packet-counter is kept with the entry in the routing table associated with each virtual circuit in each node through which the virtual circuit passes as well as at both ends of the virtual circuit. Each counter is N bits long where $2^N - 1$ is the maximum number of packets that can be lost on a node - i.e. the max. number of packets a buffer can hold for a single virtual circuit. Thus, if a message is composed of M packets and a virtual circuit creation packet has been sent previously (at which time all counters were

zeroed), the counter associated with the source process' virtual circuit is incremented once transmission of the first packet begins. The counter is then incremented with the transmission of each successive packet for all M packets.

During recovery, each virtual circuit passing through a failed node has counters on neighboring nodes on two sides of the failed node, which, if they do not match, indicate that one or more packets have been lost on the node. To integrate this check into the recovery algorithm, a node checks if it is a neighbor of the failed node when it receives a RECOVERY message. If so, the node marks the link to the failed node unusable, and then checks if any virtual circuits in it's table go through or came through, but not ending or beginning with, the failed node. For each virtual circuit which does pass through the failed node, the circuit is rerouted and a PKT_CHECK message containing the packet-counter value is sent to the destination node of the virtual circuit. This destination node will receive two such messages - one from each node on ''either side'' of the failed node. A comparison is made of the two packet-counter values and, if they do not match, a recovery coordinator is started up: RC_Z where Z is the destination process of the virtual circuit. From this point on recovery of the interacting set proceeds as described above in ''process-level recovery''. If several messages were lost on the failed node, then several interacting sets may have to be rolled back to ensure a correct system state.

6.4. Recovery in a Packet Switching Environment

In a system which does not use virtual circuits but uses packet-switching for communication we must rollback the entire system when a error is detected. This is necessary for two reasons. When communication errors are detected - i.e. two LFSR signatures do not match - we can not pinpoint when the error occured or what processes were effected by it. Secondly, if a node level error occurs we could detect if any messages in transit were lost on the node but, if any were, it could not be determined where the messages came from or where they were going. In either case, the entire system must be rolled back to ensure correct operation. Since all messages in the system are flushed and discarded during a recovery session, system-level recovery is a ''natural'' extension of node-level and will not be discussed here.

6.5. Important Details

Due to space limitations it is not feasable to describe the checkpoint and recovery algorithms in complete detail. Therefore, this section discusses some of the most important details which had to be addressed.

6.5.1. "Interfering Processes"

Since most of the system performing its normal operation while a checkpoint session is in progress, there is always the risk that a process is going to try and enter the checkpointing interacting set by sending a message to a checkpointing process. Above, we stated that once all virtual circuits whose destination is process X have been flushed, process X's state is complete and may be sent to disk. It is possible for this to *not* be the case if a message creation packet (and subsequent data packets) arrives *after* all those circuits were flushed. If the sending process is not part of the interacting set then any arriving messages can be held until X has finished checkpointing. However, if the sending process is actually part of the interacting set but has not previously communicated with process X then any arriving data packets must be included in the process X's state. To solve this problem an extra step in the checkpointing algorithm is needed. Each process' state may be sent to disk just as before, except that the process' message queue is not sent until after a CH_COMMIT message is received. The CC does not need to do this however - it saves it's process' state in one step since it knows, once it has received all expected CH_ACK messages, that the entire interacting set has been completely defined. At that time the CC sends its state to disk and also sends CH_COMMIT messages down the tree. Since it takes a "long" time to send a process state to disk, this extra step in the algorithm adds very little time to a checkpoint session.

6.5.2. Version Variables

In order to be able to recover from errors that occur or are detected during a checkpoint session, we need to be able to determine which process state to recover to - the state being saved or the state saved previously. Each process has a "version variable"[16] associated with it during a checkpoint session. When the checkpoint begins this version variable is equal to "Old Version" until the process' message queue is being sent to disk at which time it's version variable is set to "unknown". After the checkpoint handler receives a CH_RESUME message it sets its process' version variable to "1 - Old Version" which is the *new* "Old Version". Due to network delays the "unknown" time period is different depending on where a particular process resides in the recovery tree. The CC changes its version variable from Old Version to the complement of Old Version after all CH_DONE messages have arrived and before sending CH_RESUME messages, thereby signifying commitment to the new checkpoint.

During recovery, RE_ACK messages contain a copy of the sending process' version variable. If none of the processes in the interacting set have "unknown" version variables then recovery proceeds as described above - i.e. if a process' version variable is known its RH can go ahead and request the process state when it is ready to do so with no delay. If one or more but not all of the version variables are

unknown, then, as soon as a "known" version variable is encountered the RH will broadcast that version to the rest of the recovery tree. If all of the version variables are unknown then the RC concludes that failed process must have been the CC and broadcasts that the version to which recovery is to take place is Old Version.

6.5.3. Multiple Coordinators

Since errors and checkpoint timers may occur or go off at any time it is possible for two or more processes, within the same interacting set, to "simultaneously" decide to initiate a checkpoint or recovery session. To guarantee correctness of the algorithms we can not allow more than one of these sessions to run to completion. This problem can be easily solved by having a nondeterministic method to make a choice between several RCs or CCs. In order to do this, we assume that each node has a unique identifier and that there is a total ordering of these identifiers. For example, if a checkpoint handler CH_X is performing checkpoint duties for process X when another CHECKPOINT message arrives with a different CC ID attached to it CH_X will choose its CC to be the CC with the smaller ID. If this results in a change from the first CC to the second, then CH_X resends all CHECKPOINT messages it previously sent, but with the new CC ID. Any messages arriving with the usurped CC's ID are discarded, and CH_X now waits for CH_ACK messages containing the new CC ID. Once the CC has received all expected CH_ACK messages the interacting set has been determined and no other process can attempt to take control of the checkpoint session. If no change in CC takes place then any messages with the "defeated" CC ID in them are ignored.

This also takes care of the situation in node-level recovery where two, or more, failed processes are in the same interacting set. The independent recovery coordinator acts as RC for both of them, but the recovery handlers "see" two different "virtual" RCs uniquely identified by the combination of the failed process ID *and* the RC ID - i.e. it receives recovery messages from both virtual RCs. Therefore a similar deterministic choice can take place which results in the elimination of one of the virtual recovery coordinators. Thus, the "simultaneous" failure of more than one process within the same interacting set does not result in an unrecoverable situation.

7. 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 the case of a node failing where several processes and messages in transit may be lost simultaneously. These 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.

Acknowledgements

This research is supported by TRW Inc. and the State of California MICRO program. Our special thanks to Dr. R. Yanney for his continued support.

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. 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).
3. E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters* 11(1), pp. 1-4 (August 1980).
4. 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).
5. 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).
6. R. M. Fujimoto, "VLSI Communication Components for Multicomputer Networks," CS Division Report No. UCB/CSD 83/136, University of California, Berkeley, CA (1983).
7. 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).

8. D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 14-19 (July 1987).

9. 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).

10. K. H. Kim, J. H. You, and A. Abouelnaga, "A Scheme for Coordinate Execution of Independently Designed Recoverable Distributed Processes," *16th Fault-Tolerant Computing Symposium*, Vienna, Austria, pp. 130-135 (July 1986).

11. M. L. Powell and D. L. Presotto, "Publishing: A Reliable Broadcast Communication Mechanism," *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 100-109 (October 1983).

12. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2), pp. 123-165 (June 1978).

13. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1), pp. 22-33 (January 1985).

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. 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).

16. 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).

17. Y. Tamir and E. Gafni, "A Software-Based Hardware Fault Tolerance Scheme for Multicomputers," *1987 International Conference on Parallel Processing*, St. Charles, IL, pp. 117-120 (August 1987).

18. C. Whitby-Strevens, "The Transputer," *12th Annual Symposium on Computer Architecture*, Boston, MA, pp. 292-300 (June 1985).

19. W. G. Wood, "A Decentralized Recovery Control Protocol," *11th Fault-Tolerant Computing Symposium*, Portland, Main, pp. 159-164 (June 1981).