Recovery in Multicomputers with Finite Error Detection Latency*

P. Krishna N. H. Vaidya D. K. Pradhan
Computer Science Department
Texas A&M University
College Station, TX 77843-3112

Abstract

In most research on checkpointing and recovery, it has been assumed that the processor halts immediately in response to any internal failure (fail-stop model). This paper presents a recovery scheme (independent checkpointing and message logging) for a multicomputer system consisting of processors having a non-zero error detection latency. Our scheme tolerates bounded error detection latencies, thus, achieving a higher fault coverage. The simulation results show that for typical detection latency values, the recovery overhead is almost independent of the detection latency.

1 Introduction

Numerous approaches to checkpointing and rollback recovery have been proposed in the literature for fail-stop processors (e.g., [1]). While the notion of a fail-stop processor is a useful abstraction, it is expensive to implement [7]. In real systems, many error detection mechanisms have non-zero detection latency [2, 3, 4]. In this paper, we deal with a multicomputer consisting of processors whose built-in error detection mechanism can detect errors within a bounded error detection latency. These processors are named as *fail-slow* processors. A recovery scheme that tolerates a non-zero detection latency δ will be able to tolerate all faults that have a detection latency less than or equal to δ. The fault coverage increases with an increase in δ. For $\delta = 0$, the fault coverage of the scheme is equal to a scheme that assumes fail-stop operation. Depending on the fault coverage requirements, the detection latency value can be provided as an input by the system designer. This paper presents an independent checkpointing and message logging technique taking the detection latency into consideration.

A coordinated checkpointing scheme that tolerates bounded error detection latency is presented by Silva and Silva [2]. In this scheme, a processor's clock is used to estimate when an error could have occurred. However, during the detection latency period, the processor is in a spurious state, and a corrupt clock might lead to an incorrect recovery. Therefore, the scheme in [2] works only if a faulty processor's clock is always fault-free and a faulty processor timestamps messages correctly. Our approach does not depend on the processor clock for determining when an error could have occurred. Instead, the clock at the stable storage is used. Extensive simulations are carried out to evaluate the effects of error detection latency on recovery time.

This paper is organized as follows. Section 2 presents the system model. Section 3 presents some terminology and the data structures used in our scheme. Section 4 presents some definitions. Section 5 presents the recovery algorithm, Section 6 presents the simulation results, and conclusions are presented in Section 7.

2 System Model

We consider a system consisting of processes that communicate by sending messages to each other. We assume *deterministic* execution of each process in the system. Communication channels are assumed to be reliable and *FIFO*. A stable storage is provided, which is accessible to all the processes, and is unaffected by any kind of failures[1]. The stable storage also has the capability to timestamp received log entries using its local clock. We assume that there is an upper bound Δ on the message transmission time between any two processors [8]. Figure 1 illustrates operation of a *fail-slow* processor. A failure occurs at time t_F, and gets detected by the detection mechanism at time t_D after a latency period. The operating characteristics of the *fail-slow* processor are as follows : (i) $t < t_F$: Normal execution. (ii) $t_F \leq t < t_D$: Spurious mode. This is the detection latency period during which the processor may behave arbitrarily. (iii) $t \geq t_D$: The failure is detected at t_D, and all other processors are informed of the failure. Thus, all other processors will detect the failure by time $t_D + \Delta$.

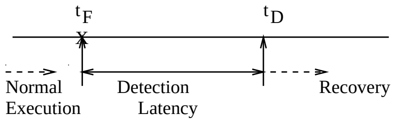

Figure 1: Operation of a *fail-slow* processor

*Research reported is supported in part by AFOSR under grant F49620-92-J-0383.

[1] Realistically, the stable storage only needs to be significantly less likely to fail as compared to rest of the system.

The proposed recovery scheme requires that the period between two adjacent checkpoints be larger than the sum of the detection latency δ and the worst case message transmission time $\triangle$. For simplicity, the proposed recovery scheme is designed to tolerate only a single process failure.

During recovery, the system must assess the damage caused by the error. The detection latency of the existent error detection mechanisms can be measured by fault-injection tools (e.g., *FERRARI* [3], *RIFLE* [4]).

After a failure has occurred, during the detection latency period, a processor may behave arbitrarily. Messages sent by the faulty process during the latency period could be contaminated and thus cause the error to be propagated to other processes. Messages received by the faulty process during the latency period could be corrupted before they are logged. Also, checkpoints taken by the faulty process during the latency period could be corrupted. Thus, the recovery algorithm should undo the damage done by the messages sent, discard the messages received, and discard the checkpoints taken, by the faulty process during the latency period.

3 Terminologies and Data Structures

CP_{ik} denotes the k-th checkpoint of process i with $k \geq 0$. CP_{ik}.time denotes the local time at the stable storage, when k-th checkpoint of process i is received by the stable storage. This timestamp is also saved in the stable storage. Send sequence number (*SSN*) of a message indicates position of the message in the sequence of outgoing messages. Receive sequence number (*RSN*) of a message indicates position of the message in the sequence of incoming messages. *Sender Log* is a log of messages sent by the process, maintained at the stable storage. For each message sent, this includes message data, destination process identification, *SSN*, and local time at the stable storage at which the message is received by the stable storage. *Receiver Log* contains information about the messages received by the process, and is maintained at the stable storage. For each message received, this includes sender process identification, *SSN* of the message, *RSN* of the message, and local time at the stable storage at which the message is received by the stable storage. For process i, the jth element of the *Max-SSN-Sent*$_i$ vector, *Max-SSN-Sent*$_{ij}$, denotes the *SSN* of the most recent message, sent by process i to process j, in the *sender log* of process i. For process i, the jth element of *Max-SSN-Recd*$_i$ vector, *Max-SSN-Recd*$_{ij}$, denotes the *SSN* of the most recent message, received by process i from process j, in the *receiver log* of process i.

The *Sender Log*, *Receiver Log*, *Max-SSN-Sent* vector, and *Max-SSN-Recd* vector must be included in the checkpoint of a process.

4 Definitions

Messages to be logged on stable storage are stored in local buffers before they are logged to the stable storage. When the buffers get full, the messages are logged by writing these buffers to the stable storage. If any message is present in the buffer during the detection latency period, the message may be corrupted. A message m' is said to "depend on" message m if message m is received by a process before it sends message m'.

Definition 1 *Let the local time at the stable storage at which a message M was logged by process i be t_M. τ is the time at the stable storage when failure of process i is detected by the stable storage. If $(\tau - \triangle - \delta) \leq t_M \leq \tau$, message m is unsafe, else the message is safe. In addition, a message dependent on an unsafe message is also unsafe.*

$(\tau - \triangle - \delta)$ gives the "pessimistic" earliest time (according to the stable storage clock) when the error could have occurred. Therefore, some uncorrupted messages that were logged before the failure actually occurred, may also be declared *unsafe*. The pessimistic behavior occurs because the stable storage clock is used to determine the *unsafe* messages. Processor clock cannot be used because it can be faulty when the processor fails (including during the latency period).

During recovery, the logs should first be made *safe*. This is done in our scheme using a procedure called *safe(τ, p)*; τ and p are the time of error detection according to the stable storage clock, and the process identifier, respectively. The *safe(τ, p)* finds the *unsafe* messages (using Definition 1) and deletes them from the logs. Thus, only *safe* messages are used during the recovery.

Definition 2 *Let τ be the local time at the stable storage when a failure of process i is detected by the stable storage. A checkpoint CP_{in} of process i is unsafe if $(\tau - \triangle - \delta) \leq CP_{in}.time \leq \tau$, otherwise, the checkpoint is safe.*

The recovery algorithm should discard an *unsafe* checkpoint, as it could be erroneous.

5 Recovery Algorithm

Message logging is performed asynchronously. Each process checkpoints independently with the constraint that time interval between two consecutive checkpoints of a process is at least $\delta + \triangle$[2].

Upon a failure, recovery is initiated to determine the recovery line to which the processes should rollback. The recovery algorithm first assesses the damage done by the latency of the detection mechanism. Basing on the knowledge of the time of error detection (according to the stable storage's clock) and δ (provided by the system designer[3]), the recovery algorithm determines the earliest time instant when the error could have occurred, and then begins the recovery.

[2] This assumption makes sure that at most one checkpoint gets affected by an error.

[3] If not, δ can be defined to be the maximum latency over all the error detection mechanisms. In those systems where the error detection mechanism can be identified, δ can be defined to be the worst case latency of that detection mechanism.

Case I : The simplest case occurs when only a checkpoint operation takes place during the error latency period, but there are no messages in the local buffer waiting to be logged on the stable storage. This case is illustrated in Figure 2. τ is the time by stable storage's clock at which the stable storage detected failure of process P. Since the transmission delay upper bound is $\triangle$, and the error detection latency is δ, the earliest instant when the error could have occurred according to the stable storage clock is $\tau - (\triangle + \delta)$. In this case, as $CP_{P2}.time > (\tau - (\delta + \triangle))$, the recovery algorithm detects that CP_{P2} is an *unsafe* checkpoint. Therefore, CP_{P2} is discarded. As shown in Figure 2, process P logged some messages at time L (i.e. the stable storage received them at L). As $L < (\tau - (\delta + \triangle))$, these messages could not have been corrupted by P's failure. Since no messages are logged by P after time L, *safe*(τ, P) does not find any *unsafe* messages. Process P rolls back to CP_{P1}. The *sender* log and *receiver* log of process P is used during the recomputation to recover process P's state before the failure. The *Max-SSN-Sent* vector inferred from the updated *sender* log is used to avoid the resending of *safe* messages. Curr_SSN$_i$ is defined to be the maximum *SSN* of the safe messages in the sender log for process i, i.e., Curr_SSN$_i$ = $\forall j\ MAX$(Max-SSN-Sent$_{ij}$). During the recomputation of the checkpoint interval, the resending of messages whose *SSN* value is less than the *Curr_SSN* is avoided. Messages are received in the same order as indicated by the *receiver* log. Since the faulty process has not propagated the error to other processes, other processes do not rollback.

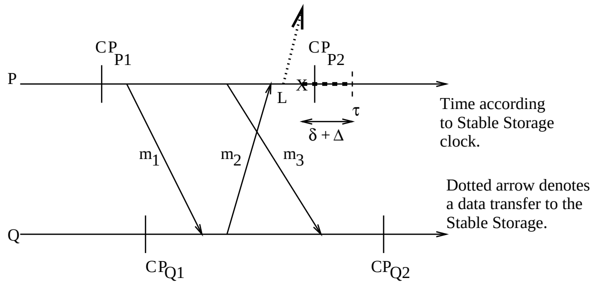

Figure 2: Unsafe Checkpoint

Case II : In this case only message transfers take place during the latency period. This case is illustrated in Figure 3. In this case, as $CP_{P1}.time < (\tau - (\delta + \triangle))$, there are no *unsafe* checkpoints. We define the phrase "logged at time t" as "received by the stable storage at time t". The logging of messages could have taken place sometime before the error had occurred, at time *L1*, or during the detection latency period at time *L2*. In both cases, the messages m_3 and m_4 will be in the local buffer during the latency period. If logging took place at *L2*, all the messages sent or received after CP_{P1} will be *unsafe*, because $L2 > (\tau - (\delta + \triangle))$. The *safe($\tau$, P)* operation will remove the entries in the logs corresponding to the *unsafe* messages, and update the *Max-SSN-Sent* and *Max-SSN-Recd* vectors. On the other hand, if logging took place at time *L1*, only m_3 and m_4 will be in the local buffer. Thus, m_3 and m_4 are *unsafe* messages. The *safe(τ, P)* operation will not detect them, as they have not been logged. As explained later, these *unsafe* messages are detected by requesting the other processes to perform a logging operation upon receipt of the *rollback request* message from P.

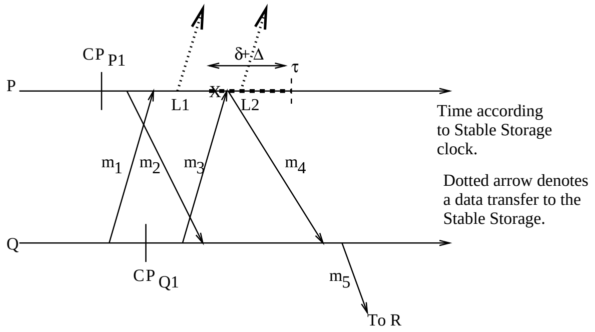

Figure 3: Unsafe Messages

Since the destination of the *unsafe* messages sent by process P may not be correctly stored in the *sender* log or the local buffer, process P sends a *rollback request* message to every other process in the system. *Max-SSN-Sent*$_{Pi}$ and *Max-SSN-Recd*$_{Pi}$, vector elements of P corresponding to a process i are tagged along with the *rollback request* message to process i. Process i, upon receipt of the *rollback request* message from P, performs a logging operation, i.e., it logs the messages present in its local buffer, and updates its *Max-SSN-Sent* and *Max-SSN-Recd* vectors.

The *Max-SSN-Sent*$_{Pi}$ value is compared with the *Max-SSN-Recd*$_{iP}$ value of process i. One of the two cases might occur:

1. *Max-SSN-Recd*$_{iP}$ = *Max-SSN-Sent*$_{Pi}$: This implies that process i did not receive any *unsafe* messages from process P. Hence, no rollback is necessary.

2. *Max-SSN-Recd*$_{iP}$ > *Max-SSN-Sent*$_{Pi}$: Inconsistency. Messages have been sent to process i by process P during the detection latency period. This will require process i to rollback to a checkpoint where the value of *Max-SSN-Recd*$_{iP}$ $\leq$ *Max-SSN-Sent*$_{Pi}$.

   The entries in the *receiver* log of process i corresponding to those messages from P whose *SSN* values are greater than *Max-SSN-Sent*$_{Pi}$ are removed. Process i starts recomputing from a previous checkpoint using the messages in its *sender* and the *receiver* log.

   If process i sends messages to any other process after it had received an *unsafe* message, those messages will also be *unsafe*. Process i sends *rollback request* message to the processes it had sent *unsafe* messages. Process i then waits for the *rollback ack* message to arrive from those processes.

For example, in Figure 3, message m_5 sent to process R, by process Q, is such an *unsafe* message, caused due to error propagation. Thus process Q will send a *rollback request* message to process R. Process Q then waits for the *rollback ack* message to arrive from process R.

Since we assume a reliable network and *FIFO* channels, the case of $\textit{Max-SSN-Recd}_{iP} < \textit{Max-SSN-Sent}_{Pi}$ will not occur.

The value of $\textit{Max-SSN-Recd}_{Pi}$ is compared with the $\textit{Max-SSN-Sent}_{iP}$ value of process i. One of the two cases might occur:

1. $\textit{Max-SSN-Recd}_{Pi} = \textit{Max-SSN-Sent}_{iP}$: This implies that process P did not receive any messages from process i during the detection latency period. Hence, there is no inconsistency.

2. $\textit{Max-SSN-Recd}_{Pi} < \textit{Max-SSN-Sent}_{iP}$: Inconsistency. Messages sent by process i have been received by process P during the latency period. This will require the process i to resend those messages whose $\textit{SSN} > \textit{Max-SSN-Recd}_{Pi}$.

Each process i sends back a *rollback ack* message back to the failed process P. A notification is also sent to process P along with the *rollback ack* message as to whether process i has to resend the messages that were discarded by process P during the *safe(τ, P)* operation. Upon receipt of *rollback ack* message from every other process, process P restarts from the previous checkpoint. It resends those messages whose *SSN* value is greater than $\textit{Curr\_SSN}_P$ to the respective processes. It requests the processes (which had tagged the notification along with the *rollback ack* message) to resend the messages.

Case III : In this case, both *unsafe* messages and *unsafe* checkpoints are present. The recovery scheme is a combination of the recovery schemes for cases I and II. The details of the algorithm are presented in [8].

6 Simulations

Simulations were performed to evaluate the effects of detection latency on recovery. Each process communicates with others by passing messages. The time at which a process sends a message is assumed to follow an exponential distribution with mean of t_m. Identity of the destination process is determined randomly (uniformly distributed). The values of the parameters used in the simulation are shown in Table 1.

Table 1: Simulation Parameters

Parameter	Value(s) chosen
Message frequency(t_m)	1 message/sec
Number of processes	10
Checkpoint interval	2, 5, 10 minutes
Checkpoint state size	100 Kbytes
Message size	2 Kbytes
Log buffer size	16, 32 Kbytes
Detection latency (δ)	0 (fail-stop) to 1 sec
Max. Comm. latency ($\triangle$)	1 ms
Disk transfer rate [5]	12.5 Mbytes/sec
Disk seek time [6]	12.5 ms
Disk rotate time [6]	13.9 ms

The performance parameters of interest are (i) *maximum* recovery overhead, (ii) *average* recovery overhead, (iii) number of processes rolled back due to error propagation and (iv) overhead due to checkpointing and logging. As stated earlier, due to detection latency, error is propagated to processes other than the failed one. Thus, during recovery, more than one process could be rolled back, and, the rollback distance could be more than one checkpoint interval. Recovery overhead for a process is defined as the period of computation lost (rollback distance) due to the failure. *Maximum* recovery overhead and *average* recovery overhead are computed as the maximum and average of recovery overheads of all the processes, respectively.

6.1 Results

The detection latency (δ) was varied from 0 to 1 sec. Zero detection latency corresponds to a fail-stop processor [7]. Typical latencies are less than 1 sec [3]. The *average* and *maximum* recovery overheads were computed for different latency values.

Figure 4 demonstrates the *average* overhead (computed as a percentage of the total runtime), for different checkpoint interval sizes, and for a log size of 16 Kb. As seen, there is an increase in the *average* recovery overhead as the latency increases. This is because, as the latency increases, the probability of *unsafe* messages being sent is high. Thus, the number of *unsafe* processes are higher. By *unsafe* processes we mean the processes which have received *unsafe* message(s). Figure 5 shows the number of processes rolled back due to possible error propagation, for different checkpoint interval sizes, and for a log size of 16 Kb. With non-zero latency value, some processes are rolled back due to possible error propagation. Figure 6 demonstrates the *maximum* overhead (computed as a percentage of the total runtime), for different checkpoint interval sizes, and for a log size of 16 Kb. As seen, the *maximum* recovery overhead is not much affected by the latency value used. The value of latency is typically small to bring about significant variation in the *maximum* recovery overhead. Since the detection latency is small compared to the checkpoint interval, the probability of a checkpoint operation taking place during error detection is small. Therefore, the *maximum* recovery overhead is relatively independent of latency. Figure 7 shows the failure-free overhead due to checkpointing and logging (computed as a percentage of the total runtime) for different checkpoint interval sizes and for log sizes of 16 Kb and 32 Kb. The overhead depends on the checkpoint size, checkpoint interval size, log size, message size and the I/O bandwidth. The failure-free overhead is quite low for realistic parameter values.

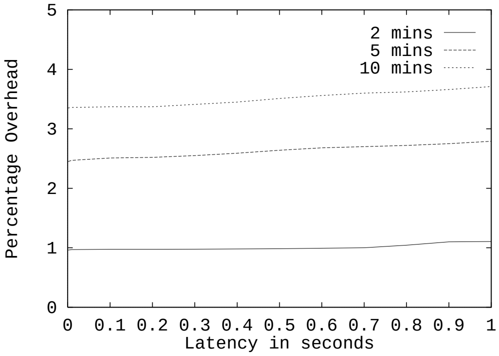

Figure 4: Average overhead

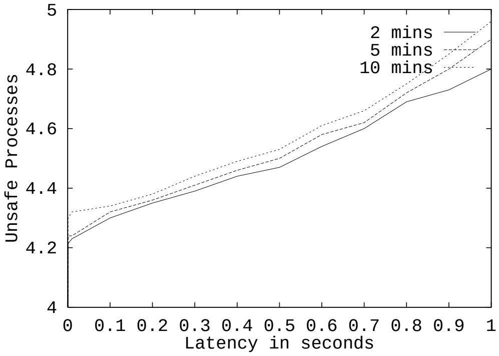

Figure 5: Unsafe Processes

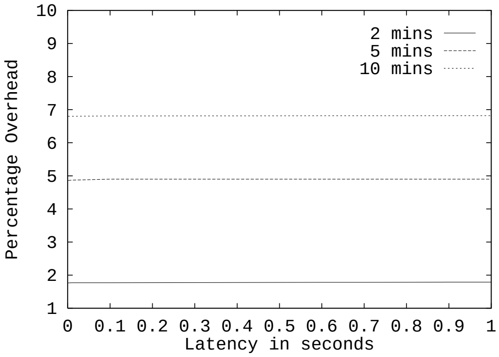

Figure 6: Maximum overhead

7 Conclusions

This paper presents an independent checkpointing and message logging scheme that tolerates error detection latency up to δ. The proposed scheme allows the system designer to provide δ as an input. Typical error detection latencies are small (less than 1 sec.), therefore δ is expected to be small. Larger the chosen value of δ, higher is the fault coverage provided by the scheme. For typical values, an increase in δ causes only a marginal increase in the *average* recovery overhead, due to possible error propagation. However, when δ is comparable with the checkpoint interval, the *maximum* and *average* recovery overhead is expected to increase. Thus, a tradeoff exists between the fault coverage and the recovery overhead. Depending on the system requirements the system designer may choose an appropriate value for δ. The simulation results show that for typical error detection latencies and checkpoint interval sizes, the proposed scheme provides a higher fault coverage (by allowing non-zero latencies) at almost no additional cost, than schemes that assume zero detection latency.

Ckp. Intl. Log Size	2 mins	5 mins	10 mins
16 Kb	4.63%	4.51%	4.49%
32 Kb	2.44%	2.35%	2.3%

Figure 7: Checkpointing and Logging overhead

References

[1] Kai Li, J. F. Naughton and J. S. Plank, “An efficient checkpointing method for multicomputers with wormhole routing,” *Intl. Journal of Parallel Programming,* Vol. 20, No. 3, pp. 159-180, June, 1992.

[2] L. M. Silva and J. G. Silva, “Global checkpointing for distributed programs,” *IEEE Symp. on Reliable Distributed Systems,* pp. 155-162, 1992.

[3] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, “FERRARI: A tool for the validation of system dependability properties,” *Proc. 23rd Int'l Symp. on Fault Tolerant Computing,* pp. 336-344, June 1992.

[4] J. Arlat et. al., “Fault injection for dependability validation: A methodology and some applications,” *IEEE Transactions on Software Engg.,* Vol. 16-2, pp. 166-182, Feb. 1990.

[5] P. Cao et. al., “The TickerTAIP parallel RAID architecture,” *Intl. Symp. on Computer Architecture,* pp. 52-63, May 1993.

[6] D. Stodolsky et. al., “Parity Logging overcoming the small write problem in redundant disk arrays,” pp. 64-75, May 1993.

[7] R. D. Schlichting and F. B. Schneider, “Fail-stop processors : An approach to designing fault-tolerant distributed computing systems,” *ACM Trans. on Computer Systems,* Vol. 1, No. 3, pp. 222-238, Aug. 1983.

[8] P. Krishna et. al., “An Optimistic Recovery Scheme for Message-Passing Multicomputer Systems with Finite Detection Latency,” Tech. Rept. 94-030, Dept. of Computer Science, Texas A&M University.