SUBMITTED TO: FTCS 23, JUNE 22-24, 1993

Progressive Retry for Software Error Recovery in Distributed Systems

Yi-Min Wang[*], *Yennun Huang*[†] and *W. Kent Fuchs*[*]

[*] Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
1308 West Main Street
University of Illinois, Urbana, IL 61801

[†] AT&T Bell Laboratories, Murray Hill, NJ 07974

Primary contact: Yi-Min Wang
E-mail: ymwang@crhc.uiuc.edu
Phone: (217) 244-7161
FAX: (217) 244-5686

Abstract

In this paper, we describe a method of execution retry for bypassing software errors based on checkpointing, rollback, message reordering and replaying. We demonstrate how rollback techniques, previously developed for transient hardware failure recovery, can also be used to recover from software faults by exploiting message reordering to bypass software errors. Our approach intentionally increases the degree of nondeterminism and the scope of rollback when a previous retry fails. Examples from our experience with telecommunications software systems illustrate the benefits of the scheme.

Key words: *distributed systems, software boundary errors, checkpointing and message logging, message reordering, progressive retry*

[1]*Acknowledgement:* This research was supported in part by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283.

1 Introduction

Numerous checkpointing and rollback recovery techniques have been proposed in the literature to recover from transient hardware failures. *Independent checkpointing* schemes [1, 2] allow maximum process autonomy and general nondeterministic execution, but suffer from potential domino effect [3]. *Coordinated checkpointing* schemes [4, 5] eliminate the domino effect by sacrificing a certain degree of process autonomy and paying the cost of extra coordination messages. Recently, a *lazy checkpoint coordination* technique [6] has been proposed as a mechanism for bounding rollback propagation and providing a flexible trade-off between run-time coordination overhead and recovery efficiency.

Log-based recovery provides another way of achieving domino-free recovery. Under the *piecewise deterministic model* [7], the domino effect is avoided through message logging and deterministic replaying. In a *pessimistic logging protocol* [8, 9], each message is logged upon receipt which prevents the rollback of a faulty process from causing the rollback of any other process. *Optimistic logging protocols* [7, 10–12] have been proposed to reduce run-time overhead by using asynchronous message logging at the expense of possible rollback propagation due to lost volatile message logs upon failure.

Instead of proposing another checkpointing and recovery protocol, this paper investigates the possibility of applying the log-based techniques to recovery from software errors [7, 13–16]. We previously proposed message reordering for changing the communication pattern at run-time in order to reduce the rollback distance for *hardware failures* [17]. In this paper, we demonstrate how message reordering can also provide an effective way of bypassing *software errors*. Fig. 1 illustrates the basic concept. When a software error is detected at the point marked "X", rollback and message replaying based on the complete checkpoint and message log information may lead to the same error. By intentionally discarding part of the message logs, we can deterministically reconstruct the system state up to the dotted line shown in

Fig. 1, and then use message reordering to introduce nondeterministic execution beyond the dotted line in order to bypass the software error. Unlike the recovery block approach [3] and the N-version programming [18] which both use *different programs* to execute on the same set of data, the above on-line retry approach [14, 19] uses the same program to operate on a *different but consistent set of data* obtained through message reordering.

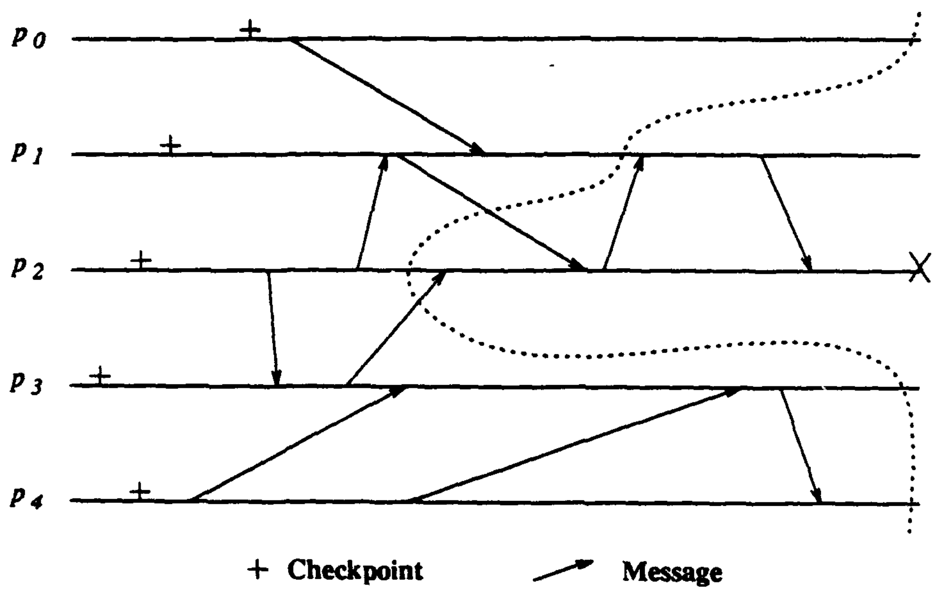


Figure 1: Nondeterministic execution beyond the dotted line through message reordering.

Based on our experience with telecommunications software systems, the technique of execution retry with rollback and message replaying has demonstrated its usefulness for bypassing the so-called *software boundary errors*. Usually, an application contains a main routine that performs the designated functions, and some *boundary code* for handling situations such as program exceptions, resource failures, urgent or unexpected messages, failures on system or function calls, etc. In many application programs that we have observed, the boundary code is usually not well tested due to the difficulty in creating such boundary conditions in a test environment [15]. Consequently, the possibility of software errors in the boundary code, called the *software boundary errors*, can potentially be higher than that in the main routine [15]. These kinds of software boundary errors may cause a catastrophic event such as the AT&T 4ESS switching systems failure on January 1990 [20]. The fact that a software boundary condition usually occurs very rarely also suggests that if a boundary error does occur, then on-line retry by replaying and/or reordering the incoming messages may be helpful in bypassing the boundary condition.

The simplest approach to execution retry is to roll back the entire system and restart from a consistent global checkpoint. This can result in nondeterministic execution in a distributed message-passing environment and this nondeterminism may result in bypassing the boundary condition. However, it is often desirable to limit the scope of rollback, the number of involved processes as well as total rollback distance, in order to achieve faster recovery [21]. It is possible that a small-scope rollback involving only a few processes suffices for successful retry. This motivates our *progressive retry* idea which progressively increases the scope of rollback to intentionally introduce more nondeterminism when a previous retry fails. Such an idea has been implemented in some telecommunication systems software and has been shown to improve the availability of the systems. The objective of this paper is to describe and formalize the concept of progressive retry with message reordering to bypass software errors and to present a systematic method for implementing the retry. The technique is being built into an existing fault tolerance library [22] in order to facilitate future software development.

2 Logical Checkpoints and Recovery Line

Let N be the number of processes in the system. Suppose p_1 in Fig. 2 initiates a rollback at the point marked "X". In a general nondeterministic execution, the rollback of p_1 to its checkpoint C will *unsend* messages M_1 and M_3, and thus require p_0 to roll back to a state before the receipt of M_1 in order to *unreceive* M_1 and similarly require p_2 to *unreceive* M_3; otherwise, M_1 and M_3 are recorded as "received but not yet sent", which results in the inconsistency of system state.

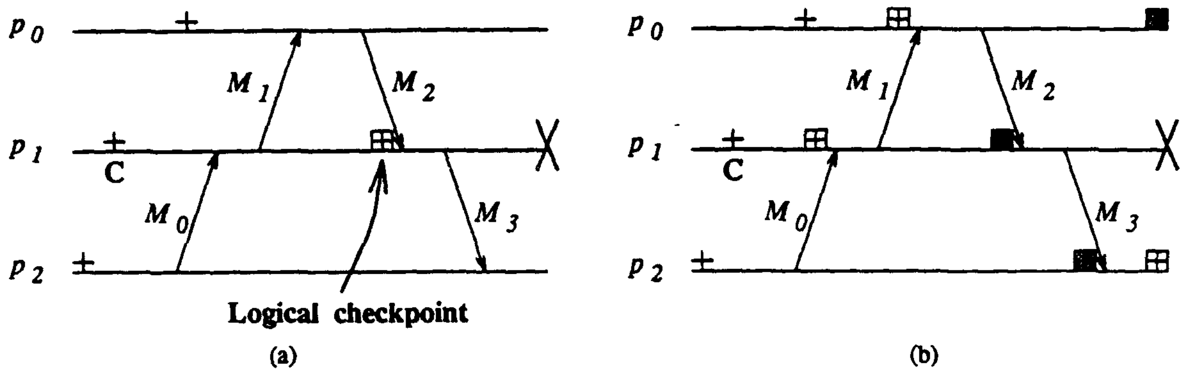


Figure 2: State consistency (a) example checkpoint and communication pattern (b) logical checkpoint dependency and recovery line.

However, if p_1 retains enough information and can guarantee to resend M_1 during its reexecution[1], the execution of p_0 based on the processing of M_1 is still valid and therefore p_0 need not roll back. This can be achieved by the *piecewise deterministic model* and additional message logging and replaying. The piecewise deterministic model says: process execution between two consecutive message receipts, called a *state interval*, is deterministic. So if p_1 has logged both the message content and the *state interval index* [10] (i.e., the processing order) for messages M_0 (but not for M_2) by the time it initiates the rollback, p_1 can deterministically reconstruct the state up to immediately before the receipt of M_2 (an nondeterministic event) and therefore M_1 remains a valid message.

A useful way to unify these two seemingly different state consistency concepts is to introduce the notion of a logical checkpoint. While a *physical checkpoint* like checkpoint C

[1]Under the fail-stop [23] assumption.

allows the restoration of process state at the point the checkpoint was taken, checkpoint C and the message log of M_0, plus the underlying piecewise deterministic model effectively place a *logical checkpoint* at the end of the state interval started by M_0 (as shown in Fig. 2(a)) because of the capability of state reconstruction. In other words, although p_1 "physically" rolls back to checkpoint C, it "logically" rolls back to the above logical checkpoint and therefore does not *unsend* M_1. For clarity, we let each physical checkpoint initiate a new state interval and represent it by a logical checkpoint at the end of that interval. Based on the above notion of logical checkpoints, the following *rollback propagation rule*[2] is then valid with or without the piecewise deterministic model:

> **if the sender rolls back and *unsends* a message M, the receiver must also roll back to *unreceive* M.**

We define a *global checkpoint* as a set of N (logical) checkpoints, one from each process. A *consistent global checkpoint* is a global checkpoint that does not contain any two checkpoints violating the above rollback propagation rule. The *recovery line* is the latest available consistent global checkpoint which uniquely minimizes the total rollback distance. As an illustration, suppose all the messages in Fig. 2(a) except for M_2 are logged when p_1 initiates the rollback. Fig. 2(b) shows the dependency graph for the available logical checkpoints. By starting with the set of the last logical checkpoints of each process and applying the rollback propagation rule described above, we can determine the recovery line to be the set of shaded checkpoints in Fig. 2(b).

[2]In contrast, when the receiver rolls back and *unreceives* a message M', the sender does not have to roll back if M' is logged and can be retrieved by the receiver during reexecution.

3 Progressive Retry for Bypassing Software Errors

We base our discussions on the following system model and recovery protocol.

FIFO channel : messages sent along the same channel between any two processes are ordered by monotonically increasing *sequence numbers*

Nondeterministic merge component [7]: messages from all incoming channels are merged by the merge component based on a possibly nondeterministic merge function, and are assigned the state interval indices

Logging before processing : every message is logged before delivery to the application process[3]

Direct dependency tracking [10, 24, 25]: only the dependency of the receiver's logical checkpoint on the sender's logical checkpoint resulting from each message processing is recorded, as opposed to the *transitive dependency tracking* which has been used in many log-based papers [7, 11].

Centralized recovery line computation : the global dependency information is collected by a single process [2, 10] which is responsible for the recovery line computation[4]

3.1 Recovery Line and Message Logs

With respect to the recovery line consisting of the shaded checkpoints shown in Fig. 3, messages can be classified into four categories.

[3]Our work can be extended to systems with asynchronous (optimistic) message logging by making additional logical checkpoints unavailable for those lost volatile message logs due to the failure.

[4]A distributed and synchronized algorithm has been proposed by Sistla and Welch [11]. A distributed and asynchronous algorithm can be found in Strom and Yemini's paper [7].

1. **Obsolete messages:** In order to reconstruct the state up to the recovery line, the system can restart from the set of *restarting checkpoints*, called the *restart line*, as illustrated in Fig. 3. Messages that were processed before the restart line, for example, M_O, are therefore obsolete messages and not useful for recovery.

2. **Messages for deterministic replay (deterministic messages):** Messages processed between the restart line and the recovery line must have both their message contents and state interval indices logged. These messages need to be replayed in their original order for deterministic state reconstruction. M_D and M'_D are such messages.

3. **In-transit (or channel-state) messages:** For messages sent before the recovery line and processed after, only the message contents in the log are valid. The state interval indices are either not logged or invalidated. Messages like M_I and M'_I belong to this category and can be processed in arbitrary order.

4. **Orphan messages:** Messages sent after the recovery line are orphan messages. M'_R can not exist because otherwise the recovery line is not consistent. M_R is invalidated by the rollback and should be discarded.

In an optimistic logging protocol [7], rollback propagation can result from the nondeterminism due to the lost volatile message logs upon failure. Based on the available message logs from stable storage, the recovery line is uniquely determined and each message must statically belong to one of the four categories depending on its position relative to the recovery line. In contrast, our retry technique progressively increases the degree of nondeterminism and the scope of rollback by discarding more message logs as a previous retry fails. At each step, a new recovery line or restart line is computed based on the remaining checkpoint and message log information. Since the recovery line moves backward in time during the progressive retry, messages belonging to the ith category can shift to the jth category, where

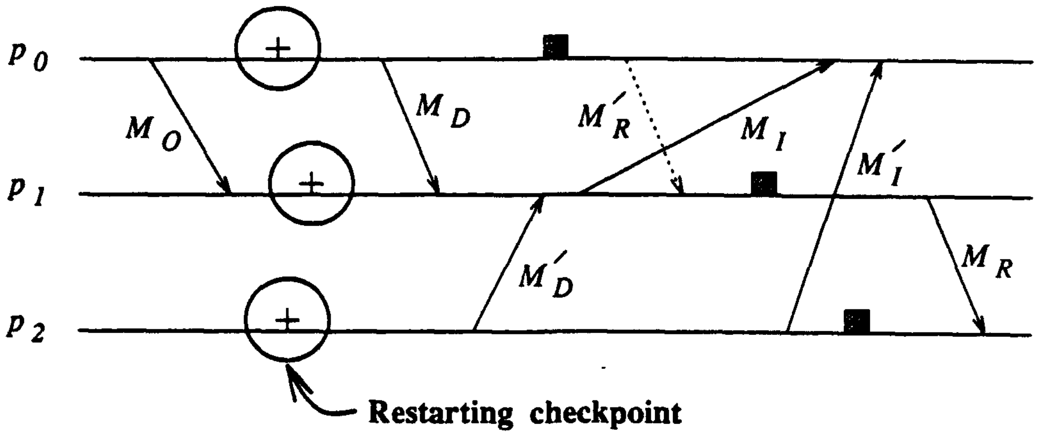


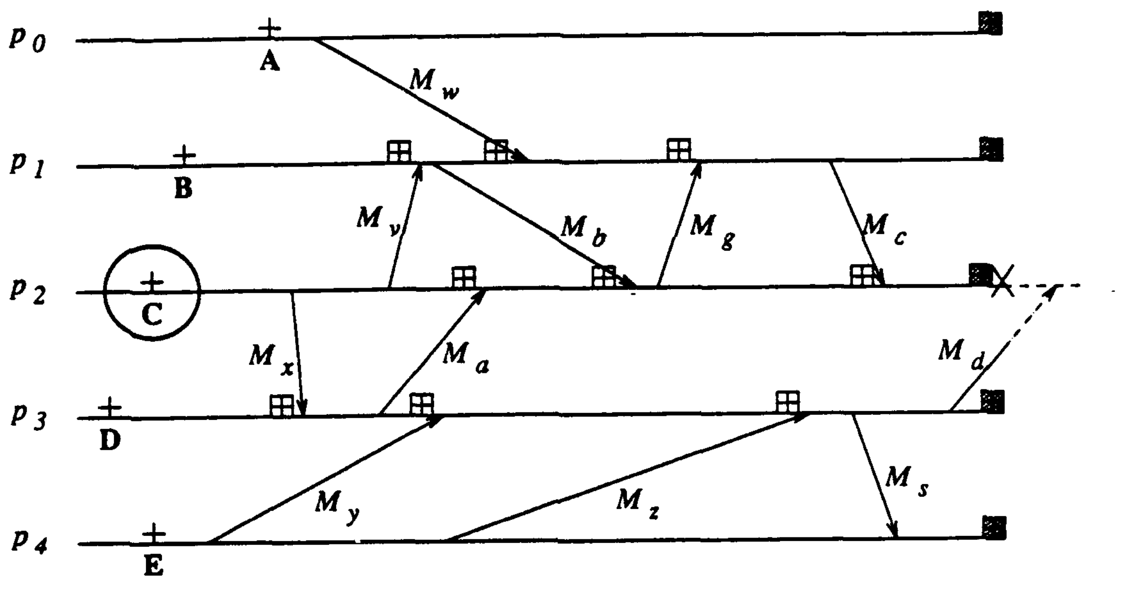
Figure 3: Example obsolete, deterministic, in-transit and orphan messages.

$j \geq i$, at a later stage.

3.2 Progressive Retry

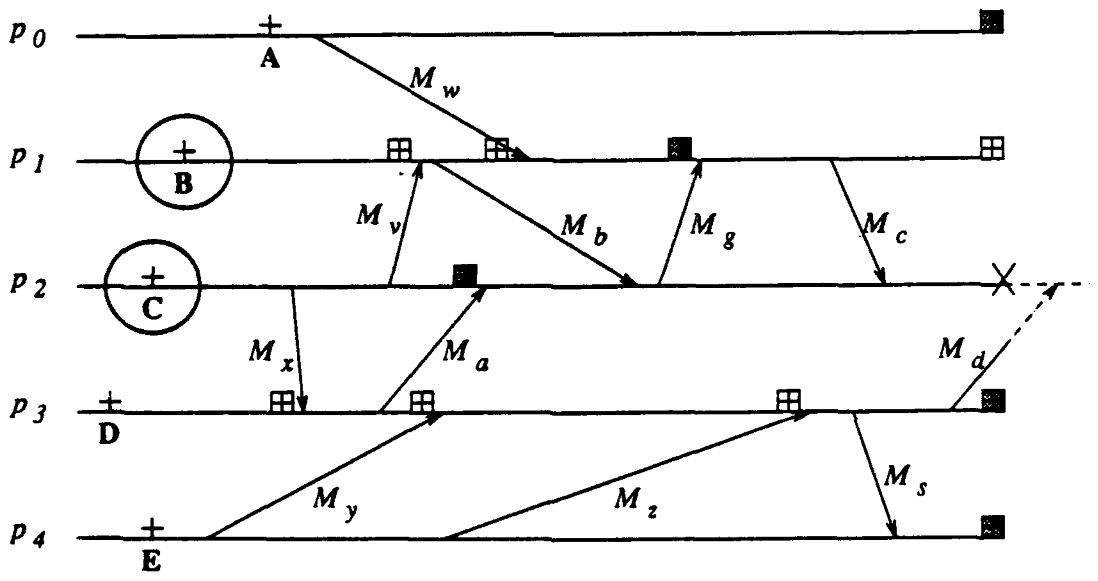
We will use the example checkpoint and communication pattern shown in Fig. 4 to illustrate progressive retry in five steps. We assume one retry per step and no hardware failure for the following discussion.

Step 1 - Receiver deterministic retry: When p_2 detects an error, it first initiates a local recovery by rolling back to checkpoint C and deterministically replaying the message logs. Because every message is logged before processing, message logs for M_a, M_b and M_c must be available and allow p_2 to reconstruct the state up to the point it detected the error, as illustrated by the recovery line shown in Fig. 4(a). In some cases, transient failures may be caused by some environmental factors which will simply disappear after the recovery, and the Step-1 retry may succeed. If the reexecution still leads to the same error, the checkpoint and message log information is copied to a trace file for off-line debugging and Step 2 is initiated.



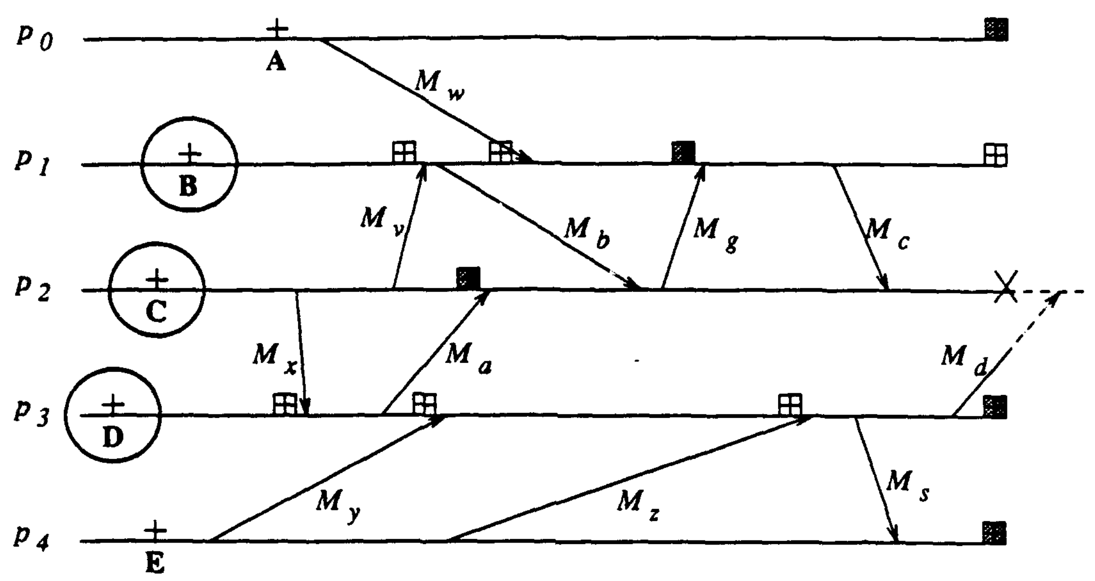
p_0
p_1
p_2
p_3
p_4
A
B
C
D
E
M_w
M_v
M_b
M_g
M_c
M_x
M_a
M_d
M_y
M_z
M_s

(a)

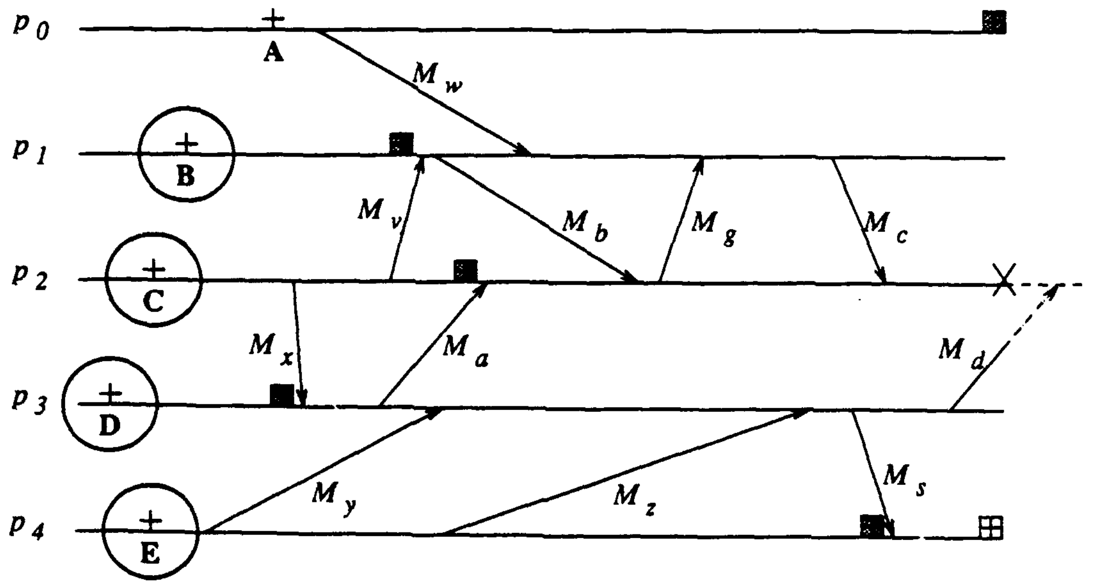


p_0
p_1
p_2
p_3
p_4
A
B
C
D
E
M_w
M_v
M_b
M_g
M_c
M_x
M_a
M_d
M_y
M_z
M_s

(b)



(c)



(d)

Figure 4: Progressive retry (a) Step 1: receiver deterministic retry (b) Step 2: receiver nondeterministic retry (c) Step 3: sender deterministic retry (d) Step 4: sender nondeterministic retry.

Step 2 - Receiver nondeterministic retry: p_2 starts introducing nondeterminism by discarding the state interval indices of M_a, M_b and M_c in order to allow message reordering. As a result, the last three logical checkpoints of p_2 are now unavailable and the resulting recovery line is shown in Fig. 4(b). Notice that only M_a and M_b are in-transit messages available for reordering; message M_c now becomes an orphan message and should be discarded.

Message reordering can be achieved by restoring the in-transit messages to the input of nondeterministic merge function and re-assigning them with possibly different state interval indices. An alternative is to group the messages from the same process together if the software bug is possibly due to concurrency control.

Step 3 - Sender deterministic retry: Messages that have been received but not yet logged, i.e., still in the system queue, can be lost upon failure. Message M_d in Fig. 4 is an example. Such lost messages can be detected[5] when the receiver receives another message from the same sender which indicates a discontinuity in the message sequence number [7]. The sender is then requested to resend the message if sender logging is available [7], or to regenerate the message through deterministic state reconstruction [11].

The immediate recovery of such lost messages is useful for increasing the number of messages available for reordering. p_2 now discards the message contents of M_a and M_b as well. Although the resulting recovery line as shown in Fig. 4(c) is the same as the one in (b), p_3 in addition to p_1 and p_2 is rolled back[6] in order to regenerate (recover) the lost message M_d. Again, if sender logging is available, p_3 can simply resend messages

[5]For some applications, lost messages may be acceptable. For example, if the lost message is a channel request message in a telephone switching application, the user will simply redial or try again later.

[6]p_2 can notify p_3 to roll back by sending p_3 the largest sequence number of any message sent from p_3 and received by p_2 before checkpoint C. Similar messages are sent to all other processes.

M_a and M_d without rolling back.

Step 4 - Sender nondeterministic retry: When reordering M_a, M_b, M_d and possibly other newly-arriving non-orphan messages still fails to bypass the software error, p_2 suspects some of these messages should not have been generated in the first place. Therefore, p_1 and p_3 are requested to roll back further by discarding the state interval indices of the message logs that can deterministically generate these messages. The resulting recovery line is given in Fig. 4(d). Nondeterminism can be introduced by p_1 reordering M_v and M_w, and p_3 reordering M_x, M_y and M_z.

Step 5 - Large-scope rollback retry: When all previous small-scope retries fail, a large-scope rollback can be initiated. Instead of backing off a few state intervals for reordering a small number of messages involving a small number of processes, all processes in the system are requested to roll back K intervals where K should be a large number compared to Step 1 through Step 4. The recovery line computed from the remaining available logical checkpoints is then used for the final-step retry.

The choice of K is a trade-off between output commit and garbage collection versus the available nondeterminism. Outputs to the outside world that cannot be rolled back should only be released after the recovery line has advanced beyond the state intervals that generate these outputs. Checkpoints and message logs can only be garbage-collected after the restart line has passed their corresponding state intervals [11]. Therefore, while a larger K means more nondeterminism is available, it also results in slower output commit and less effective garbage collection, which are translated into slower response to the users and larger space overhead, respectively. In the extreme case where fast output commit is the most important requirement for the system, only those state intervals beyond the last output can be backed off for introducing nondeterminism [7].

4 Experience and Discussion

In this section, we describe two examples from telecommunications software with software boundary errors. These errors resulted in program hang-up or program crash. However, by using the progressive retry technique (Step 2 for Case 1 and Step 3 for Case 2), these programs were able to quickly recover from the errors. To simplify the description, we have abstracted only the components which contributed to the errors. These software errors were later found and fixed. These examples show that even before the software faults were repaired, the software errors did not interrupt the services, due to progressive retry.

Case 1

In a file replication mechanism, all "open", "write" and "close" system calls are trapped by the primary node and passed to an agent process on a backup node. The agent process performs the system calls to replicate files. The agent process opens a file when an open command is received and closes a file when a close command is received. There is only one agent process to serve many applications on the primary node. Since the number of available file descriptors for the agent process is limited and each application process could open many files at the same time, the agent process may run out of file descriptors. Therefore, it has to keep track of how many files are open. A boundary condition for the agent process occurs when all file descriptors are used. The agent process then searches for an open file descriptor with the earliest access time and closes that file.

A software bug existed in the search procedure so that once the agent process entered the boundary condition, the search process never finished and the agent process hung up. The agent process implemented the checkpointing and logging mechanism and had an external hang-up detection mechanism. Once the agent process entered the boundary condition, the failure was detected and the agent process was rolled back. When the agent process was restarted, it restored the checkpointed state, reordered and reexecuted the message logs. Once the messages were reexecuted, the agent program was able to continue its operation.

The following example illustrates how progressive retry functions in this instance. Let `o1` command stand for opening file 1, `w1` command stand for writing data to file 1 and `c1` command stand for closing file 1. The agent process can open at most 2 files at the same time. The following command sequence will cause the agent program to enter the boundary condition when processing `o3` and hang up.

```
o1 o2 w1 w1 w2 w1 w2 o3 w3 c1 c2
```

Suppose the logging mechanism had logged all the commands before the failure. When the agent process is restarted, the command log may be reordered with the following sequence:

```
o1 w1 w1 w1 c1 o2 w2 w2 c2 o3 w3.
```

In this sequence, the boundary condition never occurs, and therefore the reexecution of the command log succeeds.

Case 2

In a cross-connection system, a process (BK) is used to track the available channels in the switch. The BK process gets information from two other processes: process (CA) which sends the channel allocation requests and process (DA) which sends the channel deallocation requests. A boundary condition for BK occurs when all channels are used and the process receives additional allocation requests. In that case, a clean-up procedure is called to free up some channels or to block further requests. However, the clean-up procedure contained a software glitch which could cause the process to crash.

The cross-connection system uses a daemon watcher to detect a process failure and employs checkpointing and message logging mechanism to recover from the failure. The following example illustrates how progressive retry works in this system. Suppose the number of available channels is 5. The command `r2` stands for requesting two channels, and the command `f2` stands for freeing two channels. The following command sequence can cause the BK process to crash because of the boundary error.

```
CA sends r2 r3 r1
DA sends f2 f3 f1
BK receives r2 r3 r1 and crash
```

If the message f2 is received and logged before BK crashes, BK will be able to recover by reordering the message logs. However, if BK crashes before the f2 message is logged, reordering messages r2, r3 and r1 (Step 2) will not help. In this case, the local recovery of BK fails and CA and DA will be requested to resend their messages (Step 3). Because of the nondeterminism in operating system scheduling and communication delay, the messages may arrive at BK in a different order. For example, the message order can be

```
r2 r3 f2 f3 r1 f1
```

Since the boundary error does not occur in this case, the progressive retry involving three processes succeeds.

5 Concluding Remarks

We have described a method of applying the log-based recovery technique, previously developed for fail-stop hardware failures, to recovery from transient software errors. Our five-step progressive retry approach discards partial message log information at each step in order to introduce an increasing degree of nondeterminism for bypassing software errors. Although not every software error can be recovered through message resending, reordering and replaying, we have observed that progressive retry can provide an effective way of recovering from boundary errors in long-life software systems.

The techniques described are being implemented in the fault tolerance library `libft` which has been developed in AT&T Bell Laboratories [22]. `Libft` is a C library which supports N-version programming, recovery blocks, exception handling, message logging, and checkpointing and rollback recovery, and has been used by several AT&T products. Currently, the recovery mechanism in `libft` provides the first step of receiver deterministic retry with implementation of the remaining steps in progress.

Acknowledgement

The authors wish to express their sincere thanks to Chandra Kintala (AT&T), Andy Lowry (IBM) and Pi-Yu Chung (UIUC) for their valuable discussions.

References

[1] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollback for recovery - An optimistic approach," in *Proc. IEEE Symp. on Reliable Distr. Syst.*, pp. 3–12, 1988.

[2] Y. M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems," in *Proc. IEEE Symp. on Reliable Distr. Syst.*, pp. 147–154, Oct. 1992.

[3] B. Randell, "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[4] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Proc. IEEE Symp. on Reliable Distr. Syst.*, pp. 39–47, Oct. 1992.

[6] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation." Tech. Rep. CRHC-92-26, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. Submitted to *Int'l Conf. on Distributed Computing Systems*, 1993.

[7] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. on Computer Systems*, vol. 3, pp. 204–226, Aug. 1985.

[8] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault-tolerance," in *Proc. 9th ACM Symp. on Operating Systems Principles*, pp. 90–99, 1983.

[9] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," in *Proc. 9th ACM Symp. on Operating Systems Principles*, pp. 100–109, 1983.

[10] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. of Algorithms*, vol. 11, pp. 462–491, 1990.

[11] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," in *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pp. 223–238, 1989.

[12] T. T.-Y. Juang and S. Venkatesan, "Crash recovery with little overhead," in *Proc. IEEE Int'l Conf. on Distributed Computing Systems*, pp. 454–461, 1991.

[13] J. Gray, "A census of tandem system availability between 1985 and 1990," *IEEE Trans. on Reliability*, vol. 39, pp. 409–418, Oct. 1990.

[14] J. Gray, "Dependable systems." *Keynote Speech, 11th Symp. on Reliable Distr. Syst.*, Oct. 1992.

[15] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - A study of field failures in operating systems," in *Proc. IEEE Fault-Tolerant Computing Symposium*, pp. 2–9, 1991.

[16] D. Jewett, "Integrity S2: A fault-tolerant UNIX platform," in *Proc. IEEE Fault-Tolerant Computing Symposium*, pp. 512–519, 1991.

[17] Y. M. Wang and W. K. Fuchs, "Scheduling message processing for reducing rollback propagation," in *Proc. IEEE Fault-Tolerant Computing Symposium*, pp. 204–211, July 1992.

[18] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. on Software Engineering*, vol. SE-11, pp. 1491-1501, Dec. 1985.

[19] J. Gray and D. P. Siewiorek, "High-availability computer systems," *IEEE Computer Magazine*, pp. 39–48, Sept. 1991.

[20] M. N. Meyers, "The AT&T telephone network outage of January 15, 1990." *Invited Talk at FTCS-20*, 1990.

[21] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the UNIX environment," in *Proc. Summer '92 USENIX*, pp. 31–43, June 1992.

[22] Y. Huang and C. Kintala, "Software fault tolerance: Technologies and experience." Submitted to *FTCS-23*, 1993.

[23] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. on Computer Systems*, vol. 1, pp. 222–238, Aug. 1983.

[24] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs, "Checkpoint space reclamation for independent checkpointing in message-passing systems." Tech. Rep. CRHC-92-06, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. Submitted to *IEEE Trans. on Parallel and Distributed Systems*, 1992.

[25] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent global checkpoints based on direct dependency tracking." Research Report RC 18465, IBM T.J. Watson Research Center, Yorktown Heights, New York, Oct. 1992. Submitted to *Information Processing Letters*.