

Complete Process Recovery: Using Vector Time to Handle Multiple Failures in Distributed Systems

Golden G. Richard III
Department of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: golden@cs.uno.edu

Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
Email: singhal@cis.ohio-state.edu

Abstract: *Distributed applications are generally structured as a set of communicating processes executing on multiple processors. As the number of processors in a distributed system and the running time of applications increases, the likelihood of processor failure increases. Without mechanisms for recovery, the failure of even a single processor can mandate restarting an entire application from scratch. Handling multiple failures is important because power failures, user error, etc. can often cause several machines to fail almost simultaneously. With appropriate recovery mechanisms, distributed applications can survive failures and complete restarts can be avoided.*

In this paper we present a set of distributed recovery techniques called CPR (Complete Process Recovery) which utilize vector time to handle failures and address both consistent state restoration and the associated message handling issues. The latter is important, since some recovery techniques delegate the handling of lost or duplicate messages to the message transport mechanism. This requires the ability to checkpoint the network layer (a serious restriction), since the network is generally unaware of the anomalous messages induced by process failure and recovery. CPR provides a comprehensive approach to process recovery which melds consistent state restoration and proper message handling. Our technique requires non-failed processes to roll back at most once in response to a single failure and has low message complexity. Processes required to rollback after a failure do so concurrently, which substantially decreases recovery delay after a failure has occurred.

Keywords: Distributed systems, distributed process recovery, complete process recovery, vector time, checkpointing, optimistic message logging, overlapping failures.

1 Introduction

As the number of processors in a distributed system and the running time of distributed applications executing on such a system increase, the likelihood of processor failure (for any of a number of reasons, including user error, hardware malfunction, power failure, etc.) increases. Without mechanisms for recovery, the failure of even a single processor can mandate restarting an entire application from scratch. Ideally process recovery methods should handle overlapping failures, since many distributed systems are built from clusters of machines dependent on common power and administration facilities. The failure of several processors due to localized power failure, for example, is probably far more likely than the catastrophic failure of a single processor. In the most global sense, the failure of all processors in a distributed system is an example of overlapping failures.

Rollback recovery protocols based on process checkpointing are typically used to allow distributed applications to survive failures. Two methods for taking process checkpoints are commonly used: synchronous checkpointing and asynchronous checkpointing. In *synchronous* checkpointing schemes, processes coordinate among themselves to ensure that a consistent global state always exists (e.g., [7, 8, 9]). *Asynchronous* checkpointing requires no coordination and thus no interaction between processes during checkpointing. However, extensive rollback propagation (called the *domino effect*) can occur if no globally consistent set of checkpoints exists.

Message logging is used to supplement asynchronous checkpointing and help prevent the domino effect. Message logging provides a finer-grained rollback capability than checkpointing alone since after a checkpoint has been restored, the system can be advanced to an arbitrary state by replaying a specific number of messages. Logging can be done at either the receiver or the sender. Receiver-based logging schemes fall into two broad categories, namely, pessimistic and optimistic logging. In *pessimistic* logging, incoming messages are written to stable storage before they are processed. The drawback is performance degradation, caused by the synchronization between message reception and logging to (generally slow) stable storage. *Optimistic* schemes (e.g., [15, 16]) log messages to volatile storage and periodically spool the messages to a stable log. This allows the system to log messages in batches, resulting in more efficient disk utilization. However, messages present in the volatile log at the time of a process failure are lost; consequently, other processes may have to be rolled back to establish a consistent state.

In this paper we introduce a set of efficient recovery techniques, collectively called *CPR* (Complete Process Recovery), which restore a distributed system to a strongly consistent state. Our techniques are based on vector time, asynchronous checkpointing, and optimistic message logging at the receiver. We use

optimistic receiver-based logging and asynchronous checkpointing primarily because they allow localized recovery after a failure to a larger degree than schemes based solely on sender-based logging or synchronous checkpointing. CPR requires only a single rollback for operational processes in response to a failure and directly handles duplicate, lost, and delayed messages. The latter is important, since some recovery techniques delegate the handling of lost or duplicate messages to the message transport mechanism. This requires "checkpointability" of the network layer (a serious restriction), since the network is generally unaware of the anomalous messages that can be induced by recovery. Operational processes rollback *concurrently* in response to a failure, which substantially reduces recovery delay.

The remainder of the paper is organized as follows. In Section 2, we discuss the system model. Section 3 discusses message anomalies which arise due to failure and recovery. In Section 4, we describe Complete Process Recovery (CPR). Section 5 analyzes our technique in terms of message complexity and delay and compares CPR to a related technique based on vector time [11]. Section 6 briefly discusses related work in distributed recovery. Finally, Section 7 presents conclusions and discusses work in progress. A correctness argument appears in [12].

2 System Model

The system we consider is composed of a collection of N processors, indexed 1 through N , connected via a communications network. The processors share neither a physical memory nor a common (global) clock. Processors communicate solely by exchanging messages over fault-free, FIFO communication channels. We do not assume the network layer is checkpointable as in [8], for example, which would permit the recovery scheme to dispense with handling certain message anomalies (including lost and duplicate messages) after a failure. Processors are assumed to stop functioning on a failure without generating spurious or incorrect messages. A checkpoint/restore mechanism, capable of saving and restoring the state of processes executing on the processors, is assumed to exist. Timeouts or an alternative mechanism is used to detect processor failures. Individual processes which make up an application are piecewise-deterministic, meaning their states change deterministically as a function of the messages they receive, though the application as a whole can be non-deterministic because of variable message propagation times.

3 Message Handling Issues

Synchronous and asynchronous checkpointing alone restore a distributed system to a weakly consistent state, essentially a state free from orphan messages. An *orphan* message is a message whose send event has been undone but whose receive event is recorded in the state of some process. Even with the addition of message logging, checkpointing schemes do not necessarily guarantee that anomalous messages

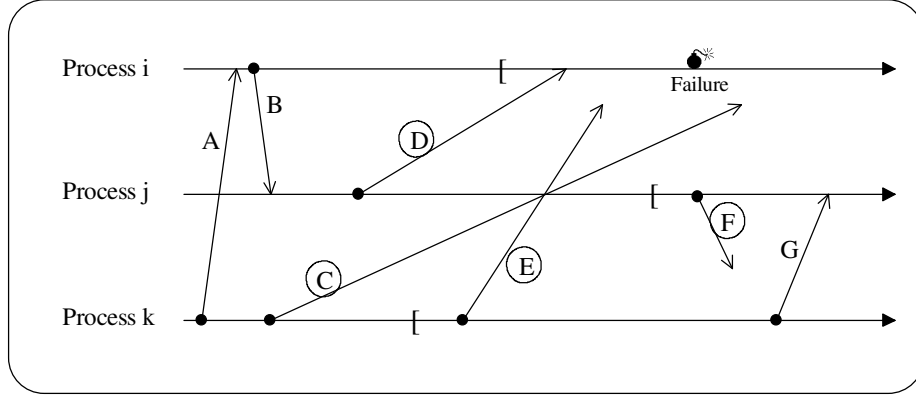


Figure 1. System state at the point of process i 's processor failure.

introduced by process recovery, such as lost, duplicate, and delayed messages, will be properly handled. Thus complete process recovery requires not only consistent state restoration, but also additional measures that ensure proper message handling. The remainder of this section examines the anomalous message types in more detail; specific techniques for dealing with them are discussed in Section 4.

In Figure 1, processes i , j , and k have sent messages A-G and messages A, B, D, and G, have been received at the points indicated. Messages C, E, and F are still in transit when the processor executing i fails. An initial attempt at recovery might restore the process states using the latest checkpoints (indicated by "[") that are available. Messages A, B, and G are handled in a correct fashion by restoration of the checkpoints, since the send and receive events for A and B are recorded and both events for G have been completely undone. We call messages A and B *normal* messages and message G a *vanished* message. Messages C, D, E, and F are potentially problematic, however. The *delayed* message C is in transit at the time of failure and several possibilities exist: C may arrive when i is still down, it may arrive when i is recovering, or it may arrive after i has completed recovery. Each of these scenarios must be dealt with correctly.

Message D is a *lost* message since the send event for D is recorded in the restored state for j but the receive event has been undone at i . Note that j will not resend D without an additional mechanism, since the send of D at j occurred before the checkpoint and the communication system successfully delivered D. Messages E and F are *delayed orphan* messages and pose perhaps the most serious problem of all; when they arrive at their respective destinations, they must be discarded since their send events have been undone. Both of these messages will be regenerated as necessary after processes resume execution from their checkpoints. Correct recovery techniques must be able to distinguish between messages like C and messages like E and F.

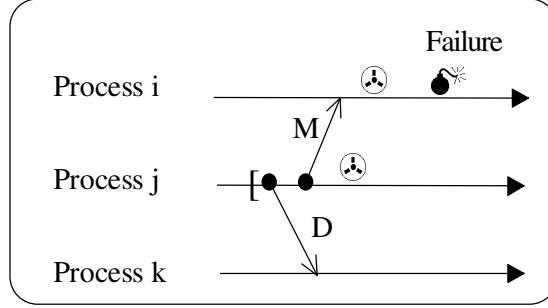


Figure 2. Duplicate messages.

The introduction of message logging and message replay during recovery can result in *duplicate* messages. In Figure 2, process j sent message D to process k and message M to process i . After process i fails, it recovers to the point indicated by the tape symbol. Assume that process j can recover to the point indicated by restoring its checkpoint and replaying its message log to achieve the system state marked by the tape symbol. During recovery, messages M and D are regenerated. Process k has received the duplicate message D twice. Note that duplicate messages occur because processes resend application messages during recovery and thus they are *not* handled automatically by the message transport layer.

4 Complete Process Recovery

CPR is based on asynchronous checkpointing with optimistic message logging at the receiver and it explicitly addresses lost messages, delayed messages, duplicate messages, etc. rather than concentrating solely on orphan message removal. For clarity, we assume that each processor executes a single process. Making each processor a *recovery unit* [16] allows this restriction to be removed easily. In the next section we examine the difficulties caused by overlapping failures. In the following sections we discuss data structures maintained by each process to support recovery, failure-free process execution, the recovery procedure, techniques for handling anomalous messages, garbage collection, and checkpoint scheduling.

4.1 Issues in Handling Simultaneous Processor Failures

Overlapping failures complicate recovery in several ways. One problem is that a process j which begins rollback/recovery in response to the failure of a process i can fail itself and develop *amnesia* with respect to i 's failure; that is, j can act in a fashion which exhibits ignorance of i 's failure. If overlapping failures are to be tolerated, a mechanism must be introduced to deal with the following type of scenario that introduces system inconsistency:

Process i fails and begins recovery by restoring a checkpoint, replaying the contents of its message log, and then transmitting recovery information to other processes. Process j receives the recovery message, immediately fails, and then begins its own recovery

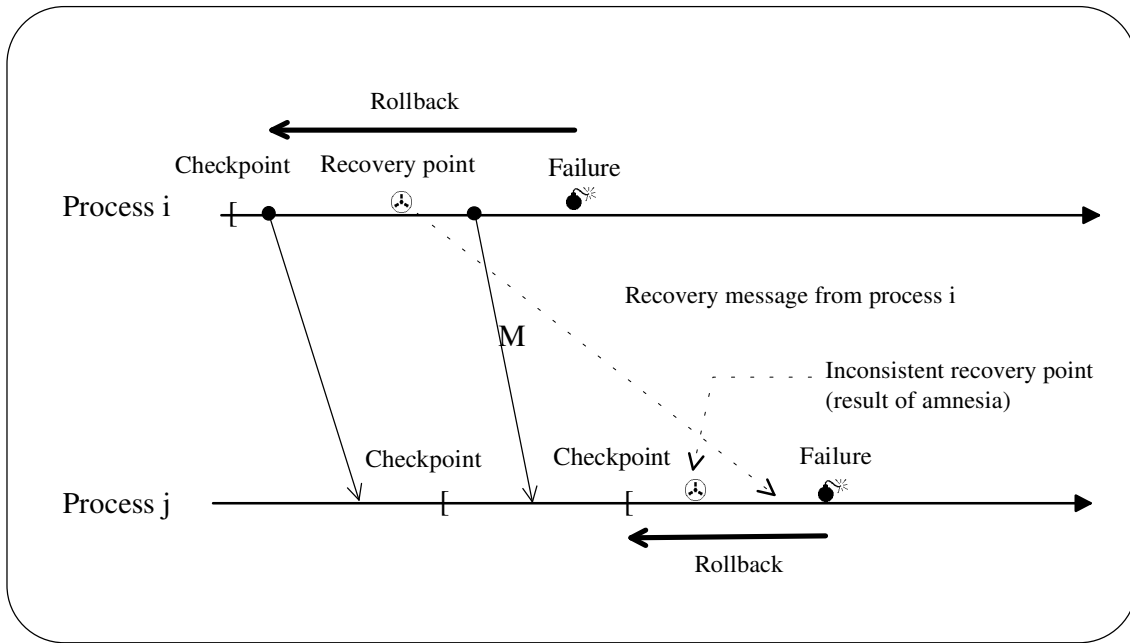


Figure 3. Amnesia during simultaneous failures.

phase by restoring its latest checkpoint, even though to avoid inconsistency, it should restore an earlier checkpoint. Process j has illustrated amnesia with respect to the recovery information sent by i . This situation is illustrated in Figure 3; the specific problem is that process j has processed the orphan message M .

Another problem is that delayed message handling is more complicated. Messages in transit which were generated by processes which rolled back may have to be discarded by the receiver. In schemes which handle only a single failure, one global incarnation number is sufficient to track which recovery period the system is currently in and thus sort out which messages should be processed and which should be rejected. A single incarnation number is insufficient in the face of overlapping failures because more than one process may simultaneously update its local value of the incarnation number. A further problem is that a failed process which is recovering may not receive cooperation from other processes to support the recovery attempt (for example, retransmission of lost messages). This is because all of the other processes are not guaranteed to remain active while recovery is underway. The techniques introduced in the remainder of Section 4 correctly address these problems.

4.2 Data Structures

Each recoverable process i maintains the following data structures:

- Volatile and stable message logs, used to log application messages.
- A vector clock [2, 10] TS_i , maintained in volatile storage, which maintains the current vector time for the process.
- An incarnation number vector INC_i , maintained in stable storage. $INC_i[i]$ is the current incarnation number for process i and $INC_i[j]$ for $i \neq j$ is the latest incarnation number for process j of which process i is currently aware.
- A direct dependency vector DV_i , maintained in volatile storage. $DV_i[j]$ is the state of process j upon which i directly causally depends and does not include transitive dependency information. DV_i is used for duplicate message detection and suppression.
- A set TIN_i containing $\langle TS, INC, j, done \rangle$ tuples, maintained on stable storage and initially empty. TS is the recovery point (a vector timestamp) for a failed process j which recovered and began execution in incarnation number INC . Variable $done$ is TRUE or FALSE depending on whether process i has completed recovery actions for this process failure. TIN_i is used in delayed message processing and to allow i to continue the recovery procedure on behalf of other failed processes should i itself fail.

4.3 Failure-free Operation under CPR

The vector timestamp TS_i is updated before being attached to a message by incrementing the i th element of TS_i and when i receives a message m as $TS_i = \text{sup}(TS_i, m.TS)$, where sup is the component-wise vector maximum operation. When a process i sends a message m , the triple $\langle TS_i, INC_i, i \rangle$ is appended to the message (notationally, $m.TS$ and $m.INC$ will be used to refer to these attachments). Differential vector techniques [14] can be used to reduce the overhead associated with attaching the vectors TS_i and INC_i to each application message.

As messages are received by a process, they are logged to volatile storage and periodically spooled to stable storage in the background. When a process fails, any messages in the volatile log are lost, but messages in the stable log are not. Periodically, processes independently take checkpoints to reduce their roll-back distance in the event of a failure.

Complete process recovery consists of two important steps: restoration of the system to a consistent state and message management (handling of duplicate, delayed, and lost messages). We first describe the

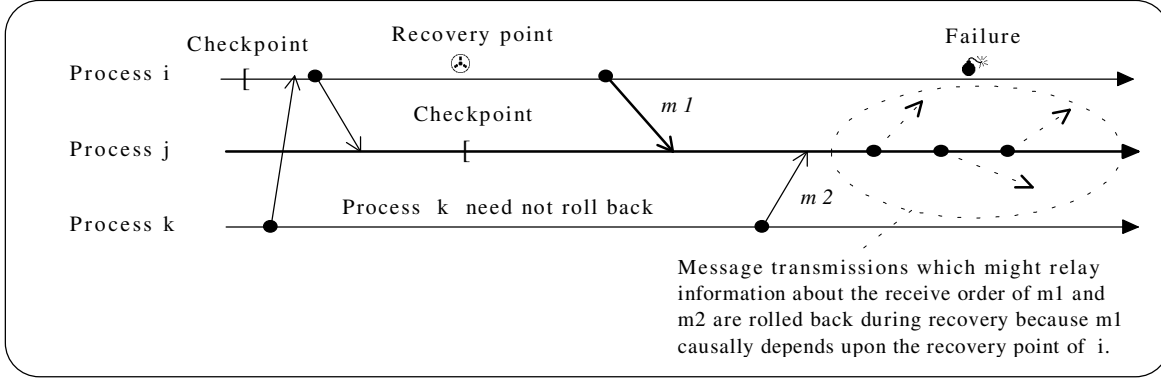


Figure 4. Orphan messages are skipped and deleted during log processing; this optimizes lost message handling and reduces the need for retransmission.

technique for restoring the system to a consistent state, which has integrated treatment of lost messages, and then discuss handling of other message types.

4.4 Restoring the System to a Consistent State

A process i performs the following steps to recover after a crash. Messages received during recovery are queued until the recovery is complete. Additional failures may occur during system recovery.

- 1 Process i determines if it was executing the recovery procedure on behalf of another process at the time of its own crash by scanning the TIN_i set for entries with $done = FALSE$. Such TIN_i entries correspond to RIM (Recovery Initiation Message) messages which had not been completely processed at the time i failed. If elements in TIN_i exist with $done = FALSE$, i finds the one with the smallest vector timestamp and assigns this timestamp to a new variable TS_{oldest} , otherwise TS_{oldest} is set to $(\infty, \infty, \dots, \infty)$. Process i restores its latest checkpoint CK (which includes the vector timestamp TS_i and direct dependency vector DV_i) from stable storage such that $CK.TS \leq TS_{oldest}$. Process i then increments its incarnation number $INC_i[i]$.
- 2 For elements $\langle TSk, INC, k, done \rangle \in TIN_i$ with $done = FALSE$, process i addresses potentially lost messages to process k by resending any messages m previously sent to k between events $TSk[i]$ and $TSi[i]$. A volatile log of messages sent to other processes, each tagged with the event number at i when the message was sent, supports this step.
- 3 Process i replays messages recorded since the restored checkpoint from its logs, skipping and deleting any messages m with $m.TS > TS_{oldest}$ (i.e., orphan messages), until the logs are exhausted. This allows orphan messages to be skipped without unnecessarily introducing lost messages. Figure 4 illustrates why orphan messages encountered during log processing can be deleted while non-

orphans are processed; in this example, any messages which relay information about the receive order of $m1$ (an orphan) and $m2$ (not an orphan) will be rolled back because they necessarily causally depend on the recovery point of the failed process. We are effectively reordering some concurrent messages and "pushing" orphans to the end of the log. Since k need not roll back, message $m2$ would become a *lost* message if log processing stopped when $m1$ was encountered.

- 4 At this point, for every $\langle TSk, INC, k, done \rangle \in TIN_i$ with $done = \text{FALSE}$, an acknowledgement is sent to process k indicating recovery with respect to this failure of k is complete and $done$ is set to **TRUE**.
- 5 Process i then broadcasts the triple $\langle TSi, INC_i[i], i \rangle$ to all other processes in the system. This message is a RIM and conveys information about the recovery point for process i to all other processes. Handshaking is required to verify that every process enters the RIM into its respective TIN set on stable storage. This means that if processors are down when a failed process attempts to recover, the failed processor will block until replacements are brought online.
- 6 Process i now awaits an acknowledgment from every other process. The application messages received and queued during recovery remain queued until all the acknowledgments have been received. When all acknowledgments have been received, process i has completed recovery and resumes normal execution.

When a process j receives a RIM $\langle TSi, INC, i \rangle$ from a process i , it performs the following steps. Application messages received during recovery are queued until recovery is complete.

- 1 $INC_j[i]$ is set to INC to update j 's view of process i 's incarnation number.
- 2 $\langle TSi, INC, i, \text{FALSE} \rangle$ is added to the set TIN_j .
- 3 If $TSi < TSj$ (i.e., j causally depends upon a state which has been rolled back), then j must roll back. Process j rolls back by restoring its latest checkpoint CK_j such that $CK_j.TS < TSi$ (i.e., the latest checkpoint for process j whose associated timestamp is not greater than the timestamp associated with the failed process' state). On the other hand, if $TSi > TSj$ or $TSi // TSj$, then j does not need to roll back because j 's state is not causally dependent on the recovery state for i .
- 4 Process j now addresses potentially lost messages to i by resending any messages m previously sent to i between events $TSi[j]$ and $TSj[j]$. A volatile log of messages sent to other processes, each tagged with the event number at j when the message was sent, can be used to support this step. Of

course increased rollback distance at j could be used to regenerate these messages, but this could be substantially more expensive.

- 5 If process j was forced to roll back in step 3, process j now replays messages recorded since the re-stored checkpoint from its logs, skipping and deleting any messages m with $m.TS > TS_i$ (i.e., orphan messages), until the log contents are exhausted.
- 6 Process j sends an acknowledgment to the failed process i to indicate it has completed recovery and then resumes execution.

4.5 Handling Overlapping Failures

A process handles RIM messages sequentially as they arrive. Unlike normal application messages, RIM messages are not queued while recovery is underway because reception of a RIM may actually preempt the recovery procedure. If a process i is already executing the recovery procedure outlined above when a RIM arrives from a failed process, three possibilities exist. The possibilities are based on a comparison between the current vector clock for process i and the timestamp associated with the arriving RIM. The possibilities and corresponding actions are as follows:

- 1 $TS_i > RIM.TS$: In this situation, the arriving RIM mandates rollback to an earlier state. The current execution of the recovery protocol is terminated and recovery is restarted based on the new RIM.
- 2 $TS_i < RIM.TS$: In this case, the RIM which initiated the current execution of the recovery protocol causally precedes the arriving RIM. No rollback in response to the arriving RIM is necessary.
- 3 $TS_i || RIM.TS$: In this situation, the RIM message conveys a recovery point which is concurrent with the state of the process i . No rollback in response to the arriving RIM is necessary but some lost messages may exist and so lost message handling must be performed as in step 4 in the section on RIM reception in Section 4.4.

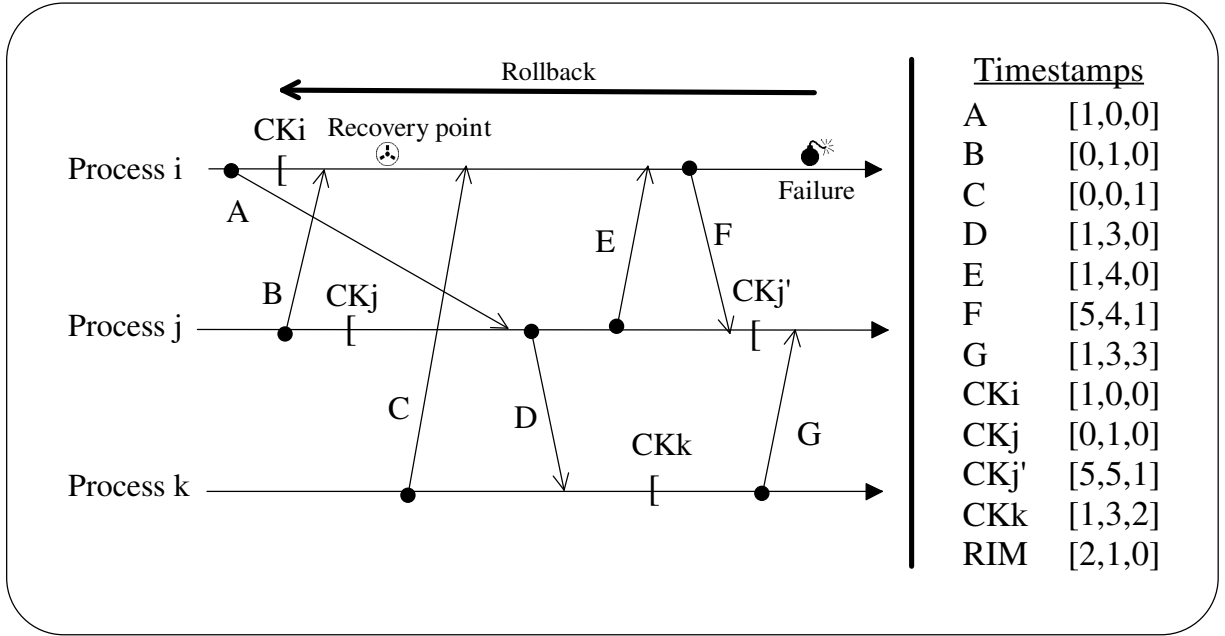


Figure 5. CPR in action.

4.6 Vanished, Delayed, and Duplicate Message Management

Unlike lost message handling, which is incorporated into the recovery procedure, handling of delayed and duplicate messages occurs as messages are received. Vanished messages require no special handling, since both their send and receive events are rolled back during recovery.

Delayed messages

Delayed messages are those whose send and receive events occur in different system incarnations. When a message m is received by a process j , $m.TS$ and $m.INC$ are examined to determine if m should be processed or discarded using the following rule:

Rule 1: Message m is discarded by process j if

$$\exists \langle TS, INC, r, flag \rangle \in TIN_j \text{ s.t. } m.TS > TS \wedge m.INC[r] < INC$$

If the condition above holds, then m was sent by a process which subsequently had to roll back to a state prior to the send of m , and thus m must be discarded.

Duplicate messages

Duplicate messages arise when a message is sent multiple times because of process rollback and message log replay. Duplicates are suppressed at the receiving end by having each process track the highest state number seen directly from every other process in the system. Vector DVi at process i tracks i 's direct

dependency on other processes [15]. DVi is initially set to all zeros. Each time process i receives a message m from a process j , $DVi[j]$ is set to the maximum of $DVi[j]$ and $m.TS[j]$. To guarantee that duplicate messages are discarded, a process uses the following rule:

Rule 2: *A message m received by process i from a process j is discarded if $m.TS[j] < DVi[j]$. Otherwise m is processed normally.*

Since vector DVi captures i 's direct dependencies on the process states of the other processes, a message received from a process with a state number lower than i has previously seen from that process is certainly a duplicate. FIFO communication channels are necessary for this rule to be applicable.

4.7 A Simple Example

We now present a simple example of CPR in action to illustrate the consistent state restoration and message handling techniques. Consider Figure 5, in which messages A through G have been sent and received at the points indicated. The vector timestamps for all message send events, recovery points, and checkpoints are provided in the diagram to clarify the example.

When process i fails, after sending message F, it has logged message B in its stable log. Messages C and E have not been logged and are lost as a result of the failure. When i begins recovery, it rolls back to its most recent checkpoint and replays messages from its stable log. In this example, only message B is replayed. Process i now sends a RIM to each of the surviving processes. The RIM messages are not shown in the diagram and incarnation numbers are ignored (since they do not contribute to this example) to reduce clutter. The RIM contains vector timestamp $[2,1,0]$, corresponding to the recovery point for process i . When process j receives the RIM from i , it must roll back to remove the orphan message F from its state. Process j rolls back to the most recent checkpoint whose vector timestamp is not greater than that of the recovery point of process i . CKj passes this test, since $CKj.TS = [0,1,0]$ and the recovery point for i has vector timestamp $[2,1,0]$. CKj' fails the test because $[5,4,1] > [2,1,0]$. Process j then replays messages from its log, skipping and deleting orphan messages. Thus messages A and G are replayed, while message F is skipped and deleted. In the process of recovery, process j resends D to process k , introducing a duplicate. The duplicate is suppressed at k using the technique described in the previous section. No lost message handling is required at process j .

While process j is recovering, process k also responds to the RIM from process i . Since process k is not causally dependent on the restored state at process i , no rollback is necessary. However, lost messages are detected, since process k sent messages to i between event counts 0 and 3 (that is, between the latest event

at k of which i is now aware and k 's current state). Message C is retransmitted to process i to correct this problem.

4.8 Garbage Collection, Output Commit, and Checkpoint Scheduling

Garbage collection falls into two categories: maintenance of the TIN_i set and maintenance of the logs. TIN_i can be garbage-collected as follows: An element $\langle TS, INC, j, done \rangle \in TIN_i$ with $done = \text{TRUE}$ can be deleted from TIN_i when a message m has been received from every process with $m.INC[j] > INC$. This signifies any messages received were sent after appropriate recovery actions corresponding to the failure of process j at vector time TS .

Since volatile (both receiver and sender) and stable logs can grow without bound and checkpoints can accumulate, it is necessary to discard obsolete log information and checkpoints to reduce storage requirements. When it can be established that the state of a process need never be rolled back prior to a checkpoint CK , all checkpoints CK' for the process with $CK'.TS < CK.TS$ can be discarded and all (received) logged messages m such that $m.TS < CK.TS$ can be discarded. By the same token, output cannot be released to the outside world unless the state in which the output is performed will never be rolled back, so this situation also requires that a process be able to establish which states are globally "permanent". A process i can be certain that it will never roll back prior to a checkpointed state with timestamp TS if for each j , process j has:

- committed a checkpoint CK to stable storage with $CK.TS[j] > TS[j]$ or
- committed a message m to stable storage which began event number s for j such that $s > TS[j]$.

Processes can periodically exchange their maximal logged states to determine when logged messages in the receive and send logs and checkpoints can be discarded.

Since taking checkpoints reduces the rollback distance should a processor subsequently fail, checkpoints should be taken as often as practical without introducing serious overhead. Many factors can be considered when scheduling checkpoints. A few of these are:

- elapsed time since the previous checkpoint,
- frequency of committing output to environments which cannot be rolled back,
- number of messages processed since the previous checkpoint,
- necessity of reducing rollback distance (and thus recovery time),
- cost of taking a checkpoint, and
- the stress placed on the network and shared file servers by processes taking checkpoints simultaneously or almost simultaneously.

Depending on the particular application environment, any or several of these factors can be taken into consideration. In environments where taking checkpoints is very expensive, for example, it may be advantageous to trade increased rollback time to reduce the runtime overhead induced by checkpointing.

5 Analysis of CPR

In this section, we compare the consistent state restoration techniques of Complete Process Recovery to related techniques proposed by Peterson and Kearns [11]. The comparison is done for a single failure. We do not discuss their message handling techniques but note in passing that lost messages of the type depicted in Figure 4 are problematic. We chose their technique because it improves on several older techniques based on asynchronous checkpointing with message logging and because, in terms of abstract cost measures, it is more directly comparable than synchronous checkpointing schemes. We chose to base our technique on asynchronous checkpointing and message logging to provide greater processor autonomy during recovery and to avoid the shared resource contention (in particular, stable storage in shared disk environments) and scalability issues which are problematic for synchronous schemes.

When a processor fails under the Peterson and Kearns recovery method, the recovering processor initiates token-passing around a logical ring of processors. The token contains the vector timestamp associated with the point of recovery for the failed processor. As each processor receives the token, it rolls back and reads messages from its log to advance its state. When log processing is complete, the processor passes the token to the next processor on the logical ring. When the token returns to the failed processor, recovery is complete. To analyze the time required to complete recovery after a failure, we propose the following parameters:

- N = the number of processors in the system.
- f = the time required by the failed processor to complete recovery to the point of beginning the token transfer.
- d = the average time required by a non-failed processor to respond to the recovery of the failed processor, including checkpoint restoration and log processing.
- d' = the maximum time required by a non-failed processor to respond to the recovery of the failed processor, including checkpoint restoration and log processing.
- m = the time required to forward recovery information from one processor to the next.
- r = time to send an acknowledgment.

Message propagation delay is included in the parameters above. In terms of these parameters, the time for recovery from a single failure in the Peterson and Kearns technique is approximately $(N - 1)(m + d) + f$,

because the token must be passed $N - 1$ times before returning to the failed processor at a cost of m per hop. Each processor other than the failed processor spends approximately d time responding to the token and the failed processor spends f time before initiating token passing.

The proposed technique uses broadcast rather than token-passing and allows processors to perform concurrent rollback/recovery in response to a failure. After advancing its state as far as possible using its latest checkpoint and the contents of its stable log, the failed processor sends recovery information to all other processes simultaneously. The recovery time for the proposed technique using the parameters above is approximately $m + d' + r + f$ if broadcast is available or $(N - 1)m + d' + r + f$ if point-to-point communication is used. Thus the proposed technique allows much faster recovery, especially if the number of processors or the recovery time for a processor (or both) are large, because rollback/recovery for non-failed processors proceeds concurrently.

6 Related Work

In this section we briefly survey related work in distributed recovery. For purposes of comparison we note that our technique requires $O(N)^1$ information to be attached to each application message and requires $O(N)$ messages to be exchanged to restore a consistent state. Differential vector techniques can substantially reduce the overhead of attaching the vector timestamp to each message (particularly when communication is localized) and thus we prefer to increase message size rather than significantly increase the number of messages transmitted and number of rollbacks during recovery. A process is forced to roll back at most once in response to a single failure in the system and all non-failed processes roll back concurrently. Overlapping failures are supported.

Sistla and Welch's algorithm [15] requires $O(N^2)$ message exchanges and $O(N)$ information to be attached to each message. The algorithm tolerates only non-overlapping failures and requires processes to roll back multiple times in response to a failure. In addition, lost messages are not handled directly. Strom and Yemini's optimistic recovery algorithm [16] attaches $O(N)$ information to each message and potentially requires a large number of rollbacks ($O(2^N)$) per failure, but does handle overlapping failures.

Juang and Venkatesan's technique [6] attaches little information to the application messages but requires $O(N * E)$ messages during recovery where E is the number of communication links in the system. Their technique requires N rounds of process rollback after a failure. Johnson and Zwaenpoel [4] present an optimistic message logging and checkpointing method based on a central server. Johnson [5] introduces a technique which allows processes to commit states to stable storage as needed to ensure fast output to the

¹ N is the number of processes in the system

outside world. Elnozahy and Zwaenepoel [1] discuss the Manetho rollback recovery protocol, based on *antecedence graphs*. An antecedence graph is attached to each application message and tracks the causal history of the message. When a process fails and recovers, it utilizes the antecedence graph and communicates with other processes in order to regenerate a consistent state.

Synchronous checkpointing schemes were proposed by Koo and Toueg [8], Kim and Park [7], Leu and Bhargava [9], Silva and Silva [13], Venkatesh et al. [18], Wang and Fuchs [20], and Wojcik and Wojcik [19]. The synchronous checkpointing schemes typically handle overlapping failures, but also require more time for output commit, since a round of checkpoints must be taken before output is delivered to environments which cannot be rolled back. The degree to which a process can recover independently after a failure is also diminished when compared to logging techniques, because any processes with which the failed process has communicated since the last round of checkpoints must be rolled back.

7 Conclusions and Work in Progress

We have presented a set of techniques for process recovery, collectively named Complete Process Recovery, which restore a distributed system to a strongly consistent state. The techniques correctly handle the unorthodox message states which can arise due to processor failure and recovery, including duplicate, lost, and delayed messages. Our techniques assume reliable message delivery but do not assume that the network layer is checkpointable. Overlapping failures are handled correctly, which is especially important in environments where problems such as local power failures, user error, or administrative decisions cause the simultaneous failure of multiple processors involved in supporting a distributed application.

After a single failure, non-failed processors in the system roll back at most once and recover *concurrently* as necessary. Our analysis shows the technique has substantially lower message complexity and recovery-time delay than many other checkpointing/logging-based recovery techniques. CPR allows faster output commit and provides greater potential for local recovery than consistent checkpointing schemes. Our log processing method reorders the logs to eliminate orphan messages and thus eliminates the need to retransmit some lost messages.

We are currently developing a testbed for experimental evaluation of CPR and other recovery techniques. The testbed is based on PVM (Parallel Virtual Machine) [3] and supports synchronous and asynchronous checkpointing, optimistic and pessimistic message logging, and efficient vector time facilities for both the static and dynamic process creation paradigms. Our goal is to move beyond abstract cost measures and limited simulation and evaluate process recovery mechanisms thoroughly. Of particular interest are:

- Time overhead imposed by the recovery technique in the absence of failures
- Time overhead during recovery
- Storage overheads, both for local/shared disk and local memory
- Storage contention issues
- Communication overhead, both during normal execution and during recovery periods
- Scalability

This tool will allow recovery techniques to be evaluated under real application loads and will provide information on the system overhead incurred both during recovery *and* in periods where the system is failure-free. This is a large-scale project and is currently under construction at the University of New Orleans.

Acknowledgements

Thanks to Kevin Cutress, a graduate student at the University of New Orleans, for helpful questions and suggestions which improved the clarity of this paper.

References

- [1] E. N. Elnozahy, W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit," *IEEE Trans. on Computers*, vol. 41, no. 5, pp. 526-531, May 1992.
- [2] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," *Australian Computer Science Communications*, vol. 10, no. 1, pp. 56-66, February 1988.
- [3] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, V. S. Sunderam, *Parallel Virtual Machine- A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.
- [4] D. B. Johnson, W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms*, No. 11, pp. 462-491, 1990.
- [5] D. B. Johnson, "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 86-95, October 1993.
- [6] T. Y. Juang, S. Venkatesan, "Crash Recovery With Little Overhead," *IEEE 11th Intl. Conf. on Distributed Computing Systems*, 1991.
- [7] J. L. Kim, T. Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 8, pp. 955-960, August 1993.
- [8] R. Koo, S. Toueg, "Checkpointing and Rollback Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 1, pp. 23-31, January 1987.
- [9] P-J. Leu, B. Bhargava, "Concurrent Robust Checkpointing and Recovery in Distributed Systems," *Proc. of the 4th Intl. Symp. on Data Engineering*, Los Angeles, CA, pp. 154-163, February 1988.
- [10] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, pp. 215-226, North-Holland, 1989,

- [11] S. L. Peterson, P. Kearns, "Rollback Based on Vector Time," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 68-77, October 1993.
- [12] G. G. Richard III, M. Singhal, "Complete Process Recovery: Using Vector Time to Handle Multiple Failures in Distributed Systems," Technical Report UNOCS-TR95-02, University of New Orleans, 1995.
- [13] L. Silva, J. Silva, "Global Checkpointing for Distributed Programs," *Proc. 11th Symp. on Reliable Distributed Systems*, pp. 155-162, October 1992.
- [14] M. Singhal, A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, pp. 47-52, August 1992.
- [15] A. P. Sistla, J. L. Welch, "Efficient Distributed Recovery Using Message Logging," *Proc. of the 8th Annual ACM Symp. on Principles of Distributed Computing*, August 1989.
- [16] R. E. Strom, S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. on Computer Systems*, vol. 3, no. 3, pp. 204-226, August 1985.
- [17] Z. Tong, R. Kain, W. Tsai, "Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 246-251, March 1992.
- [18] K. Venkatesh, T. Radhakrishnan, H. F. Li, "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery," *Information Processing Letters*, pp. 295-303, July 1987.
- [19] Z. Wojcik, B. Wojcik, "Fault Tolerant Distributed Computing Using Atomic Send-Receive Checkpoints," *Proc. of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 215-222, 1990.
- [20] Y-M. Wang, W. K. Fuchs, "Lazy Checkpoint Coordination for Bounding Rollback Propagation," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 78-85, October 1993.