CONCURRENT CHECKPOINTING AND RECOVERY IN DISTRIBUTED SYSTEMS
A. Gerald Naveen and Dr. S. Mercy Shalinie

Department of Computer Science and Engineering,
Thiagarajar College of Engineering, Madurai- 625 015 Tamil Nadu India
(E-mail: ageraldnaveen@yahoo.com ; shalinie_m@yahoo.com)

ABSTRACT
The main objective of this paper is to speed up the consistent state restoration of distributed systems. Process recovery uses vector time to address unusual message handling issues and overlapping failures. Single rollback of non-failed process in response to a single failure has low message complexity. After a failure, processes required to rollback do so concurrently, which substantially decreases recovery delay.

1.1 INTRODUCTION
Distributed computing applications involve a set of communicating processes executing on multiple processors. The probability of processor failure increases as the number of processors and the running time of distributed applications executing in a distributed system becomes more. A single processor’s failure could mandate the restart of an entire application from scratch without any recovery mechanisms. The failure of multiple processors, due to such factors as power failures, user error and hardware malfunction, can be even more likely than single processor failure.

There are some recovery techniques that delegate the handling of lost or duplicate messages to the message-transport mechanism[1]-[2]. It requires the ability to checkpoint the network layer (a serious restriction), because the network is generally unaware of the anomalous messages induced by process failure and recovery. Process Recovery[3]-[5], a set of distributed recovery that melds consistent-state restoration and proper message handling and can handle overlapping failures. The technique used in this paper requires only a single rollback of non-failed processes in response to a single failure and has low message complexity. The processes required to rollback do so concurrently after a failure, which substantially decreases recovery delay.

1.2 THE SYSTEM MODEL
The system model considered comprises of a collection of N processors, indexed 1 through N, connected through a communications network. The processors do not share a physical memory or a common clock. The processors communicate solely by exchanging messages over fault-free, FIFO communication channels. It is not assumed that the network layer is checkpointable[2]. It is assumed that, in a failure, processors stop functioning without generating spurious or incorrect messages, and that a checkpoint/restore mechanism, capable of saving and restoring the state of processes executing, exists. Timeouts, or an alternative mechanism, detect processor failures. The application’s individual processes are piecewise-deterministic, meaning their stages change deterministically as a function of the messages they receive, although the application as a whole can be non-deterministic because of variable, message-propagation times.

1.3 MESSAGE-HANDLING ISSUES
Introducing Synchronous and asynchronous checkpoints alone can restore a distributed system to a weakly consistent state – essentially, a state free from orphan messages (messages with their send events undone but with their receive events recorded in the state of some process). But even with the addition of message logging, checkpointing schemes do not guarantee that anomalous messages 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[6]-[8].

For example, in Fig.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, using the latest available checkpoints (indicated by |) might restore the process states. Restoration of the checkpoints correctly handles messages A, B and G because the send and receive events for A and B are recorded and both events for G have been completely undone. The messages A and B are called normal messages and message G as a vanished message.

Messages C, D, E and F are potentially problematic, however. In transit at the time of failure, the delayed message C has several possibilities: C might arrive before i recovers, it might arrive while i is recovering or it might arrive after i has completed recovery. Each of these scenarios must be dealt with correctly.

Message D, with the send event recorded in the restored state for j, but with its receive event undone at i, is a lost message. Note that j will not resend D without and additional mechanism, because j sent D 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 because their send events have been undone. Processes, after resuming execution from their checkpoints, will regenerate both of these messages as necessary. Correct recovery techniques must distinguish between messages like C and those like E and F.

During recovery, the introduction of message logging and replay can result in duplicate messages. In 2, process j sent message D to process k and message M to process i. After process i fails, it recovers to the recovery point. Assume that process j can recover to the indicated point by restoring its checkpoint and replaying its message log to achieve the system. During recovery, process j regenerates messages M and D. Process k has received the duplicate message D twice. Duplicate messages occur because processes resend application messages during recovery, and this the message transport layer does not automatically handle the problem.

1.4 PROCESS RECOVERY
Process recovery restores a distributed system to a strongly consistent state. The technique used in this paper is based on vector time, asynchronous checkpointing, and optimistic message logging at the receiver. Optimistic, receiver-based logging and asynchronous checkpointing primarily is used because they allow a larger degree of localized recovery after a failure than schemes based solely on sender-based logging or synchronous checkpointing. PR requires only a single rollback for operational processes in response to a failure and directly handles duplicate, lost and delayed message removal. For clarity, it is assumed that each processor executes a single process. Making each processor a recovery unit allows this restriction to be removed easily.

1.5 HANDLING SIMULTANEOUS PROCESSOR FAILURES
Overlapping failures complicate recovery in several ways. For one thing, a process j that 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 that exhibits ignorance of i’s failure. For example, in Fig.3, 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 phase by restoring its latest
checkpoint, even though to avoid inconsistency, it should restore an earlier checkpoint. The specific problem is that process j has processed the orphan message M. Process j has illustrated amnesia with respect to the recovery information sent by i. If overlapping failures are to be tolerated, a mechanism must be introduced to deal with amnesia and the resulting system inconsistencies.

Overlapping failures also make handling delayed messages more complicated. The receiver might have to discard the messages in transit generated by processes that rolled back. In schemes that handle a single failure, one global incarnation number is sufficient to track system’s current recovery period and, thus sort out which messages to process and to reject. With overlapping failures, a single incarnation number is insufficient because more than one process might simultaneously update its incarnation number’s local value. Furthermore, a recovering failed process might not receive cooperation from other processes to support its recovery attempt. This is because the other processes are not guaranteed to remain active while the first failed process recovers.

1.6 FAILURE FREE OPERATION

Under normal operation, whenever a process receives a message, the CPR appends the message with the Timestamp and the incarnation number. Differential vector techniques can reduce the overhead associated with attaching the vectors $TS_i$ and $INC_i$ to each application message.

As a process received messages, it logs them to volatile storage. Use of volatile storage is to increase the speed of logging. But because it is volatile, the messages are written in blocks to stable storage in the background periodically. When a process fails, the messages in the volatile storage are lost, but messages in the stable log are not. Periodically, processes independently take checkpoints to reduce their rollback distance in the event of a failure.

2 RESTORING THE SYSTEM TO A CONSISTENT STATE

A process i performs the following steps to recover after a crash. The process queues the messages received during recovery until the recovery is complete. Additional failures can 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. If elements in $TIN_i$ exist with done=False, i finds the one with the smallest vector time stamp and assigns this time stamp to a new variable $TS_{\text{oldest}}$; otherwise its latest checkpoint $CK$ from stable storage such that $CK.TS \leq TS_{\text{oldest}}$. Process i then increments its incarnation number $INC[i]$. 

2) For elements $<TS_k, INC, k,\text{done}>$ belongs to $TIN_i$ with done=False, process i addresses potentially lost messages to process k by resending any messages m previously sent to k between events $TS_k[i]$ and $TS_i[i]$. 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 from its logs messages recorded since the restored checkpoint, skipping and deleting any messages m with $m.TS > TS_{\text{oldest}}$, until the logs are exhausted. This allows the process to skip orphan messages without introducing lost messages.

4) Any messages that relay information about the receive order of an orphan and not an orphan will be rolled back because they depend causally on the failed process’s recovery point. By doing this, some concurrent messages are reordered and orphans are pushed to the end of the log. Because k need not roll back, message B would become a
lost message if log processing stopped when A was encountered.

5) At this point, for every \(<TS_k, INC_k, done>\) belonging to TIN, with done=False, an acknowledgement is sent to process k indicating recovery is complete, with respect to k’s failure, and done is set to True.

6) Process i then broadcasts the triple \(<TS_i, INC_i[i], i>\) to all other processes in the system. This RIM message 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 come online.

7) Process i now awaits an acknowledgement from all other processes. The application messages received and queued during recovery remain queued until the process receives all the acknowledgments. Upon their receipt, process i has completed recovery and resumes normal execution.

3 COMPARISON STRATEGY

The consistent state restoration techniques is compared with techniques proposed by Sandy Peterson and Phil Kearns[9]. The comparison is made for single failure. This method is chosen because it improves on several older techniques based on asynchronous checkpointing with message logging. The techniques used in this paper is based on asynchronous checkpointing and message logging to provide greater processor autonomy during recovery and to avoid the shared resource contention and scalability issues that are problematic for synchronous schemes.

When the processor fails under the Peterson and Kearns recovery method, the recovering processor initiates token passing around a logical ring of processors. To analyze the time required to complete recovery after a failure, the following parameters are considered:

1) \(N=\) the number of processors in the system

2) \(F=\) the time required by the failed processor to complete recovery to the point of beginning the token transfer

3) \(d=\) the average time required by a nonfailed processor to respond to the recovery of the failed processor, including checkpoint restoration and log processing.

4) \(d'=\) the maximum time required by a nonfailed processor to respond to the recovery of the failed processor, including checkpoint restoration and log processing.

5) \(m=\) the time requires to forward recovery information from one processor to the next

6) \(r=\) the time to send an acknowledgement

Message propagation delay is included in the parameters given 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.

Process recovery uses broadcast rather than token passing and allows processors to perform concurrent rollback and recovery in response to a failure. After advancing its state as far as possible using its latest checkpoint and its stable log’s contents, the failed processor simultaneously sends recovery information to all other processes. The recovery time using the parameters above is approximately \(m+d'+r+f\) if broadcast is available or \((N-1)\)

\(m+d'+r+f\) if it uses point-to-point
communication. Thus, faster recovery is obtained, especially if the number of processors or the recovery time for a processor is large, because rollback and recovery for no failed processors proceed concurrently.

4 CONCLUSION
Usage of vector time is a different approach for handling multiple failures in a distributed system. It speeds up the consistent-state restoration of distributed systems. This is a better solution over Complete Process Recovery wherein the entire process is rolled back. Vector Time approach even supports multiple failures at the same time. Other features include time overhead imposed by the recovery technique in the absence of failures and scalability.

5 REFERENCES


