FAULT TOLERANT MECHANISMS FOR EFFICIENT DATA RECOVERY IN GRID ENVIRONMENT

Saritha.G, MTech(SE)

Abstract: ***Large clusters, high availability clusters and grid deployments often suffer from network, node or operating system faults and thus require the use of fault tolerant programming models. Distributed systems today are ubiquitous and enable many applications, including client-server systems, transaction processing, World Wide Web, and scientific computing, among many others. The vast computing potential of these systems is often hampered by their susceptibility to failures. Therefore, many techniques have been developed to add reliability and high availability to distributed systems. This paper presents two such techniques: Checkpointing Based Rollback and Log Based Rollback which allows efficient recovery in dynamic heterogeneous system as well as multithreaded applications.***

Keywords: grid computing, fault tolerance, rollback recovery, checkpointing, event logging.

INTRODUCTION

Rollback recovery treats a distributed system as a collection of application processes that communicate through a network. Fault tolerance is achieved by periodically using stable storage to save the processes' states during failure-free execution. Upon a failure, a failed process restarts from one of its saved states, thereby reducing the amount of lost computation. Each of the saved states is called a checkpoint.

Rollback recovery has many flavors. For example, a system may rely on the application to decide when and what to save on stable storage. Or, it may provide the application programmer with linguistic constructs to structure the application. We focus in this survey on transparent techniques, which do not require any intervention on the part of the application or the programmer. The system automatically takes checkpoints according to some specified policy, and recovers automatically from failures if they occur. This approach has the advantages of relieving the application programmers from the complex and error-prone chores of implementing fault tolerance and of offering fault tolerance to existing applications written without consideration to reliability concerns.

Rollback recovery has been studied in various forms and in connection with many fields of research. Thus, it is perhaps impossible to provide an extensive coverage of all the issues related to rollback recovery within the scope of one article. This survey concentrates on the definitions, fundamental concepts, and implementation issues of rollback-recovery protocols in distributed systems. The coverage excludes the use of rollback recovery in many related fields such hardware-level instruction retry, distributed shared memory, real-time systems, and debugging. The coverage also excludes the issues of using rollback recovery when failures could include Byzantine modes or are not restricted to the fail-stop model. Also excluded are rollback-recovery techniques that rely on special language constructs such as recovery blocks and transactions.

Message-passing systems complicate rollback recovery because messages induce inter-process dependencies during failure-free operation. Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called rollback propagation . To see why rollback propagation occurs, consider the situation where a sender of a message m rolls back to a state that precedes the sending of m . The receiver of m must also roll back to a state that precedes m 's receipt; otherwise, the states of the two processes would be inconsistent because they would show that message m was received without being sent, which is impossible in any correct failure-free execution. Under some scenarios, rollback propagation may extend back to the initial state of the computation, losing all the work performed before a failure. This situation is known as the domino effect [4].
The domino effect may occur if each process takes its checkpoints independently—an approach known as independent or uncoordinated checkpointing . It is obviously desirable to avoid the domino effect and

therefore several techniques have been developed to prevent it. One such technique is to perform coordinated checkpointing in which processes coordinate their checkpoints in order to save a system-wide consistent state [8]. This consistent set of checkpoints can then be used to bound rollback propagation. Alternatively, communication-induced checkpointing forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes. Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

These approaches discussed above implement checkpoint-based rollback recovery, which relies only on checkpoints to achieve fault-tolerance. In contrast, log-based rollback recovery combines checkpointing with logging of nondeterministic events. Log-based rollback recovery relies on the piecewise deterministic (PWD) assumption [2], which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's determinant [4]. By logging and replaying the nondeterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed. Log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints. It is therefore particularly attractive for applications that frequently interact with the outside world , which consists of all input and output devices that cannot roll back. Log-based rollback recovery has three flavors, depending on how the determinants are logged to stable storage. In pessimistic logging , the application has to block waiting for the determinant of each nondeterministic event to be stored on stable storage before the effects of that event can be seen by other processes or the outside world. Pessimistic logging simplifies recovery but hurts failure-free performance. In optimistic logging, the application does not block, and determinants are spooled to stable storage asynchronously. Optimistic logging reduces the failure-free overhead, but complicates recovery. Finally, in causal logging, low failure-free overhead and simpler recovery are combined by striking a balance between optimistic and pessimistic logging.

2. RELATED WORK

2.1 System Model:

A message-passing system consists of a fixed number of processes that communicate only through messages. Throughout this survey, we use N to denote the total number of processes in a system. Processes cooperate to execute a distributed application program and interact with the outside world by receiving and sending input and output messages, respectively. Figure 1 shows a sample system consisting of three processes, where horizontal lines extending toward the right-hand side represent the execution of each process, and arrows between processes represent messages.

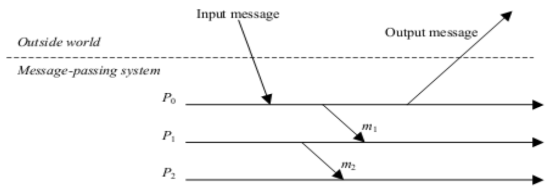

Figure 1. An example of a message-passing system with three processes.

2.2 Consistent system states

A global state of a message-passing system is a collection of the individual states of all participating processes and of the states of the communication channels. Intuitively, a consistent global state is one that may occur during a failure-free, correct execution of a distributed computation. More precisely, a consistent system state is one in which if a process's state reflects a message receipt, then the state of the corresponding sender reflects sending that message. For example, Figure 2 shows two examples of global states—a consistent state in Figure 2(a), and an inconsistent state in Figure 2(b). Note that the consistent state in Figure 2(a) shows message m1 to have been sent but not yet received. This state is consistent, because it represents a situation in which the message has left the sender and is still traveling across the network. On the other hand, the state in Figure 2(b) is inconsistent because process P2 is shown to have received m2 but the state of process P1 does not reflect sending it. Such a state is impossible in any failure-free, correct computation.

Inconsistent states occur because of failures. For example, the situation shown in part (b) of Figure 2 may occur if process P1 fails after sending message m2 to P2 and then restarts at the state shown in the figure.A fundamental goal of any rollback-recovery protocol is to bring the system into a consistent state when inconsistencies occur because of a failure. The reconstructed consistent state is not necessarily one that has occurred before the failure. It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free, correct execution.

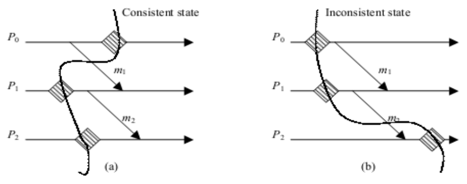

Figure 2. An example of a consistent and inconsistent state.

2.3 In-Transit Messages

In Figure 2(a), the global state shows that message m1 has been sent but not yet received. We call such a message an in-transit message. When in-transit messages are part of a global system state, these messages do not cause any inconsistency. However, depending on whether the system model assumes reliable communication channels, rollback-recovery protocols may have to guarantee the delivery of in-transit messages when failures occur. For example, the rollback-recovery protocol in Figure 3(a) assumes reliable communications, and therefore it must be implemented on top of a reliable communication protocol layer. In contrast, the rollback-recovery protocol in Figure 3(b) does not assume reliable communications.

Reliable communication protocols ensure the reliability of message delivery during failure-free executions. They cannot, however, ensure by themselves the reliability of message delivery in the presence of process failures. For instance, if an in-transit message is lost because the intended receiver has failed, conventional communication protocols will generate a timeout and inform the sender that the message cannot be delivered. In a rollback-recovery system, however, the receiver will eventually recover. Therefore, the system must mask the timeout from the application program at the sender process and must make in-transit messages available to the intended receiver process after it recovers, in order to ensure a consistent view of the reliable system . On the other hand, if a system

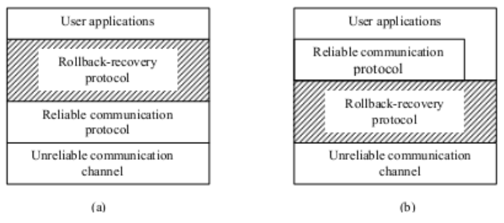
User applications
Rollback-recovery protocol
Reliable communication protocol
Unreliable communication channel
(a)
User applications
Reliable communication protocol
Rollback-recovery protocol
Unreliable communication channel
(b)

Figure3:
Implementation of rollback-recovery (a) on top of a reliable communication protocol; (b) directly on top of unreliable communication channels.

model assumes unreliable communication channels, as in Figure 3(b), then the recovery protocol need not handle in-transit messages in any special way. Indeed, lost in-transit messages because of process failures cannot be distinguished from those caused by communication failures in an unreliable communication channel. Therefore, the loss of in-transit messages due to either communication or process failure is an event that can occur in any failure-free, correct execution of the system.

2.4 Checkpointing Protocols and the Domino Effect

In checkpointing protocols, each process periodically saves its state on stable storage. The saved state contains sufficient information to restart process execution. A consistent global checkpoint is a set of N local checkpoints, one from each process, forming a consistent system state. Any consistent global checkpoint can be used to restart process execution upon a failure. Nevertheless, it is desirable to minimize the amount of lost work by restoring the system to the most recent consistent global checkpoint, which is called the recovery line [4][5]. Processes may coordinate their checkpoints to form consistent states, or may take checkpoints independently and search for a consistent state during recovery out of the set of saved individual checkpoints. The second style, however, can lead to the domino effect [4]. For example, Figure 4 shows an execution in which processes take their checkpoints—represented by black bars—without coordinating with each other. Each process starts its execution with an initial checkpoint. Suppose process P2 fails and rolls back to checkpoint C. The rollback "invalidates" the sending of message m6, and so P1 must roll back to checkpoint B to "invalidate" the receipt of that message. Thus, the invalidation of message m 6 propagates the rollback of process P2 to process P1, which in turn "invalidates" message m7 and forces P0 to roll back as well.

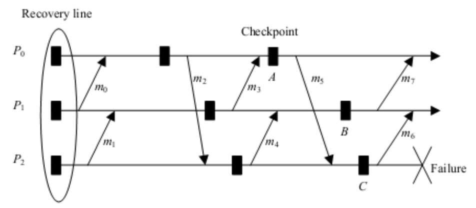

Figure4:
Rollback propagation, recovery line and the domino effect.

This cascaded rollback may continue and eventually may lead to the domino effect, which causes the system to roll back to the beginning of the computation, in spite of all the saved checkpoints. In the example shown in Figure 5, cascading rollbacks due to the single failure of process P2 may result in a recovery line that consists of the initial set of checkpoints, effectively causing the loss of all the work done by all processes. To avoid the domino effect, processes need either to coordinate their checkpoints so that the recovery line is advanced as new checkpoints are taken, or to combine checkpointing with event logging.

2.5 Logging Protocols:

Log-based rollback recovery uses checkpointing and logging to enable processes to replay their execution after a failure beyond the most recent checkpoint. This is useful when interactions with the outside world are frequent, since it enables a process to repeat its execution and be consistent with output sent to the outside world without having to take expensive checkpoints before sending such output. Additionally, log-based recovery generally is not susceptible to the domino effect, thereby allowing processes to use uncoordinated checkpointing if desired.

Log-based recovery relies on the piecewise deterministic (PWD) assumption. Under this assumption, the rollback recovery protocol can identify all the nondeterministic events executed by each process, and for each such event, logs a determinant that contains all information necessary to replay the event should it be necessary during recovery.

A state interval is recoverable if there is sufficient information to replay the execution up to that state interval despite any future failures in the system. Also, a state interval is stable if the determinant of the nondeterministic event that started it is logged on stable storage [1]. A recoverable state interval is always stable, but the opposite is not always true [8].

Figure 5 shows an execution in which the only nondeterministic events are message deliveries. Suppose that processes P 1 and P2 fail before logging the determinants corresponding to the deliveries of m6 and m5, respectively, while all other determinants survive the failure. Message m7 becomes an orphan message because process P2 cannot guarantee the regeneration of the same m6 during recovery, and P1 cannot guarantee the regeneration of the same m7 without the original m6. As a result, the surviving process P0 becomes an orphan process and is forced to roll back

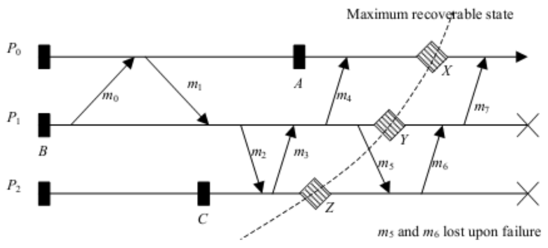

Figure 5: Message logging for deterministic replay.

as well. States X , Y and Z form the maximum recoverable state [8], i.e., the most recent recoverable consistent system state. Processes P0 and P2 roll back to checkpoints A and C, respectively, and replay the deliveries of messages m4 and m2, respectively, to reach states X and Z. Process P1 rolls back to checkpoint B and replays the deliveries of m1 and m3 in their original order to reach state Y. During recovery, log-based rollback-recovery protocols force the execution of the system to be identical to the one that occurred before the failure, up to the maximum recoverable state. Therefore, the system always recovers to a state that is consistent with the input and output interactions that occurred up to the maximum recoverable state.

2.6 Checkpoint-Based Rollback Recovery

Upon a failure, checkpoint-based rollback recovery restores the system state to the most recent consistent set of checkpoints, i.e. the recovery line [4][5]. It does not rely on the PWD assumption, and so does not need to detect, log, or replay nondeterministic events. Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback recovery. But checkpoint-based rollback recovery does not guarantee that pre-failure execution can be deterministically regenerated after a rollback. Therefore, checkpoint-based rollback recovery is ill suited for applications that require frequent interactions with the outside world, since such interactions require that the observable behavior of the system during recovery be the same as during failure-free operation.

Checkpoint-based rollback-recovery techniques can be classified into three categories: uncoordinated checkpointing , coordinated checkpointing , and communication-induced checkpointing. We examine each category in detail.

2.6.1 Uncoordinated Checkpointing

Uncoordinated checkpointing allows each process maximum autonomy in deciding when to take checkpoints. The main advantage of this autonomy is that each process may take a checkpoint when it is most convenient. For example, a process may reduce the overhead by taking checkpoints when the amount of state information to be saved is small [5]. But there are several disadvantages. First, there is the possibility of the domino effect, which may cause the loss of a large amount of useful work, possibly all the way back to the beginning of the computation. Second, a process may take a useless checkpoint that will never be part of a global consistent state. Useless checkpoints are undesirable because they incur overhead and do not contribute to advancing the recovery line. Third, uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to invoke periodically a garbage collection algorithm to reclaim the checkpoints that are no longer useful. Fourth, it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

2.6.2 Coordinated Checkpointing

Coordinated checkpointing requires processes to orchestrate their checkpoints in order to form a consistent global state. Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint. Also, coordinated checkpointing requires each process to maintain only one permanent checkpoint on stable storage, reducing storage overhead and eliminating the need for garbage collection. Its main disadvantage, however, is the large latency involved in committing output, since a global checkpoint is needed before output can be committed to the outside world.

A straightforward approach to coordinated checkpointing is to block communications while the checkpointing protocol executes [2]. A coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives this message, it stops its execution, flushes all the communication channels, takes a *tentative* checkpoint, and sends an acknowledgment message back to the coordinator. After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol. After receiving the commit message, each process removes the old permanent checkpoint and atomically makes the *tentative* checkpoint permanent. The process is then free to resume execution and exchange messages with other processes. This straightforward approach, however, can result in large overhead, and therefore non-blocking checkpointing schemes are preferable [8][7].

2.6.3 Communication-induced Checkpointing

Communication-induced checkpointing avoids the domino effect while allowing processes to take some of their checkpoints independently [8]. However, process independence is constrained to guarantee the eventual progress of the recovery line, and therefore processes may be forced to take additional checkpoints. The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints. Communication-induced checkpointing piggybacks protocol-related information on each application message. The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line. The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring high latency and overhead. It is therefore desirable in these systems to reduce the number of forced checkpoints to the extent possible. In contrast with coordinated checkpointing, no special coordination messages are exchanged.

2.7 Log-Based Rollback Recovery

As opposed to checkpoint-based rollback recovery, log-based rollback recovery makes explicit use of the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event [7]. Such an event can be the receipt of a message from another process or an event internal to the process.

Log-based rollback-recovery protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process, i.e., a process whose state depends on a nondeterministic event that cannot be reproduced during recovery. The way in which a specific protocol implements this condition affects the protocol's failure-free performance overhead, latency of output commit, and simplicity of recovery and garbage collection, as well as its potential for rolling back correct processes. There are three flavors of these protocols:

- Pessimistic log-based rollback-recovery protocols guarantee that orphans are never created due to a failure. These protocols simplify recovery, garbage collection and output commit, at the expense of higher failure-free performance overhead.
- Optimistic log-based rollback-recovery protocols reduce the failure-free performance overhead, but allow orphans to be created due to failures. The possibility of having orphans complicates recovery, garbage collection and output commit.
- Causal log-based rollback-recovery protocols attempt to combine the advantages of low performance overhead and fast output commit, but they may require complex recovery and garbage collection.

2.8 Combining Log-Based Recovery with Coordinated Checkpointing

Log-based recovery has been traditionally presented as a mechanism to allow the use of *uncoordinated* checkpointing with no domino effect. But a system may also combine event logging with coordinated checkpointing, yielding several benefits with respect to performance and simplicity [2][4]. These benefits include those of coordinated checkpointing—such as the simplicity of recovery and garbage collection—and those of logbased recovery—such as fast output commit. Most prominently, this combination obviates the need for flushing the volatile message logs to stable storage in a sender-based logging implementation. Thus, there is no need for maintaining large logs on stable storage, resulting lower performance overhead and simpler implementations. The combination of coordinated checkpointing and message logging has been shown to outperform one with uncoordinated checkpointing and message logging [2]. Therefore, the purpose of logging should no longer be to allow uncoordinated checkpointing. Rather, it should be the desire for enabling fast output commit for those applications that need this feature.

3 Conclusion

We have reviewed and compared different approaches to rollback recovery with respect to a set of properties including the assumption of piecewise determinism, performance overhead, storage overhead, ease of output commit, ease of garbage collection, ease of recovery, freedom from domino effect, freedom from orphan processes, and the extent of rollback. These approaches fall into two broad categories: checkpointing protocols and log-based recovery protocols.
Checkpointing protocols require the processes to take periodic checkpoints with varying degrees of coordination. At one end of the spectrum, coordinated checkpointing requires the processes to coordinate their checkpoints to form global consistent system states. Coordinated checkpointing generally simplifies recovery and garbage collection, and yields good performance in practice. At the other end of the spectrum, uncoordinated checkpointing does not require the processes to coordinate their checkpoints, but it suffers from potential domino effect, complicates recovery, and still requires coordination to perform output commit or garbage collection.
Between these two ends are communication-induced checkpointing schemes that depend on the communication patterns of the applications to trigger checkpoints. These schemes do not suffer from the domino effect and do not require coordination. Recent studies, however, have shown that the nondeterministic nature of these protocols complicates garbage collection and degrades performance.

Log-based rollback recovery based on the *PWD* assumption is often a natural choice for applications that frequently interact with the outside world. Log-based recovery allows efficient output commit, and has three flavors, pessimistic, optimistic, and causal.

The simplicity of pessimistic logging makes it attractive for practical applications if a high failure-free overhead can be tolerated. This form of logging simplifies recovery, output commit, and protect surviving processes from having to roll back. These advantages have made pessimistic logging attractive in commercial environment where simplicity and robustness are necessary. Causal logging can be employed to reduce the overhead while still preserving the properties of fast output commit and orphan-free recovery. Alternatively, optimistic logging reduces the overhead further at the expense of complicating recovery and increasing the extent of rollback upon a failure.

REFERENCES:

[1] L. Sarmenta, “Sabotage-Tolerance Mechanisms for Volunteer Computing Systems,” Future Generation Computer Systems, vol. 18, no. 4, 2002.

[2] L. Alvisi and K. Marzullo, “Message Logging: Pessimistic, Optimistic, Causal and Optimal,” IEEE Trans. Software Eng., vol. 24, no. 2, pp. 149-159, Feb. 1998.

[3] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth, “Solving Large Quadratic Assignment Problems on Computational Grids,” Math. Programming, vol. 91, no. 3, 2002.

[4] R. Baldoni, “A Communication-Induced Checkpointing Protocol That Ensures Rollback-Dependency Trackability,” Proc. 27th Int’l Symp. Fault-Tolerant Computing (FTCS ’97), p. 68, 1997

[5] G. Stellner, “CoCheck: Checkpointing and Process Migration for MPI,” Proc. 10th Int’l Parallel Processing Symp. (IPPS ’96), pp. 526-531, Apr. 1996

[6] F. Galile´e, J.-L. Roch, G. Cavalheiro, and M. Doreille, “Athapascan- 1: On-Line Building Data Flow Graph in a Parallel Language,” Proc. Seventh Int’l Conf. Parallel Architectures and Compilation Techniques (PACT ’98), pp. 88-95, 1998.

[7] D.K. Pradhan, Fault-Tolerant Computer System Design. Prentice Hall, 1996.

[8] Samir Jafar, Axel Krings, Senior Member, IEEE, and Thierry Gautier. Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing, In IEEE transactions on dependable and secure computing, vol. 6, no. 1, january-march 2009