CHAPTER 12

RECOVERY

12.1 INTRODUCTION

Recovery in computer systems refers to restoring a system to its normal operational state. Recovery may be as simple as restarting a failed computer or restarting failed processes. However, from the following discussion, it will be clear that recovery is generally a very complicated process.

In general, resources are allocated to executing processes in a computer. For example, a process has memory allocated to it and a process may have locked shared resources, such as files and memory. Under such circumstances, if a process fails, it is imperative that the resources allocated to the failed process be reclaimed so that they can be allocated to other processes. If a failed process has modified a database, then it is important that all the modifications made to the database by the failed process are undone. On the other hand, if a process has executed for some time before failing, it would be preferable to restart the process from the point of its failure and resume its execution. By restarting from the point of failure, the situation of having to reexecute the process from the beginning is avoided, which may be a time consuming and expensive operation.

Distributed systems provide enhanced performance and increased availability (see Sec. 4.2). One way of realizing enhanced performance is through the concurrent execution of many processes, which cooperate in performing a task. If one or more of the cooperating processes fail, then the effects due to the interactions of the failed processes with the other processes must be undone, or every failed process would have to restart

from an appropriate state. Increased availability in distributed systems is realized mainly through replication (e.g., data, processes, and hardware components can be replicated). If a site fails, copies of data stored at that site may miss updates, thus becoming inconsistent with the rest of the system when it becomes operational. Recovery in such cases involves the question of how not to expose the system to data inconsistencies and bring back the failed site to an up-to-date state consistent with the rest of the system.

In this chapter, (1) the basic causes that lead to failures and the types of failures that occur in a computer system are introduced. (2) The question of how a process can recover from failure when it does not interact with another process is discussed. (3) The effects of a process failing on other processes in concurrent systems, and techniques to recover cooperating processes without them having to resume execution from the beginning, are described. (4) Finally, recovery in distributed database systems is discussed.

12.2 BASIC CONCEPTS

A *system* consists of a set of hardware and software components and is designed to provide a specified service. The components of a system may themselves be systems together with interrelationships [32]. *Failure* of a system occurs when the system does not perform its services in the manner specified [32]. An *erroneous state* of the system is a state which could lead to a system failure by a sequence of valid state transitions [22]. A *fault* is an anomalous physical condition. The causes of a fault include design errors (such as errors in system specification or implementation), manufacturing problems, damage fatigue or other deterioration, and external disturbances (such as harsh environmental conditions, electromagnetic interference, unanticipated inputs or system misuse) [27]. An *error* is that part of the system state which differs from its intended value [32].

From the above definitions, it can be seen that *an error is a manifestation of a fault in a system, which could lead to system failure* (Fig. 12.1). Therefore, to recover from a system failure, we need to rid the system state of errors. In other words, *failure recovery* is a process that involves restoring an erroneous state to an error-free state.

12.3 CLASSIFICATION OF FAILURES

Failures in a computer system can be classified as follows:

PROCESS FAILURE. In a process failure, the computation results in an incorrect outcome, the process causes the system state to deviate from specifications, the process may fail to progress, etc. Examples of errors causing processes to fail are deadlocks, timeouts, protection violation, wrong input provided by a user, consistency violations (which can happen if an optimistic concurrency control technique is employed). Depending on the type of the error causing a process to fail, a failed process may be aborted or restarted from a prior state. For example, a deadlocked process can be restarted from a prior state, where it can try to acquire the resources again. On the other hand, a wrong input in the initial stages may require a process to be aborted. In this chapter, we do

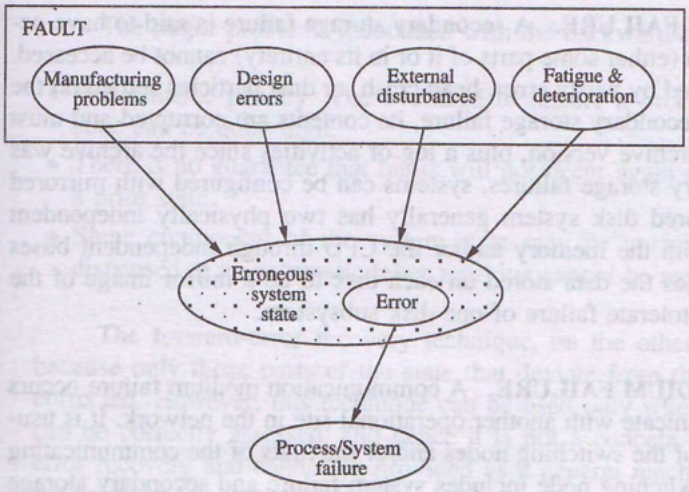

FIGURE 12.1
An error is a manifestation of fault and can lead to failure.

not consider process failures where processes behave maliciously. A discussion on that type of process behavior can be found in Chap. 8.

SYSTEM FAILURE. A system failure occurs when the processor fails to execute. It is caused by software errors and hardware problems (such as CPU failure, main memory failure, bus failure, power failure, etc.). In the case of a system failure, the system is stopped and restarted in a correct state. The correct state may be some predefined state or a prior state (checkpoint) of the system saved on nonvolatile storage. In this chapter, we assume that systems behave as fail-stop processors [33]. These type of systems have very simple failure mode operating characteristics. The only visible effects of a failure in such a system are: the system stops executing; and the internal state and contents of the volatile storage belonging to the system are lost.

A system failure can further be classified as follows [9].

- An *amnesia* failure occurs when a system restarts in a predefined state that does not depend upon the state of the system before its failure.
- A *partial-amnesia* failure occurs when a system restarts in a state wherein a part of the state is the same as the state before the failure and the rest of the state is predefined, i.e., it does not depend upon the state of the system before its failure. This type of failure typically occurs in file servers when a file server crashes and restarts, or when a system is restarted from a checkpoint.
- A *pause* failure occurs when a system restarts in the same state it was in before the failure.
- A *halting* failure occurs when a crashed system never restarts.

SECONDARY STORAGE FAILURE. A secondary storage failure is said to have occurred when the stored data (either some parts of it or in its entirety) cannot be accessed. This failure is usually caused by parity error, head crash, or dust particles settled on the medium. In the case of a secondary storage failure, its contents are corrupted and must be reconstructed from an archive version, plus a log of activities since the archive was taken. To tolerate secondary storage failures, systems can be configured with mirrored disk systems [3]. A mirrored disk system generally has two physically independent disks that communicate with the memory and/or the CPU through independent buses and controllers. This enables the data stored on each disk to be a mirror image of the other. Thus, a system can tolerate failure of one disk subsystem.

COMMUNICATION MEDIUM FAILURE. A communication medium failure occurs when a site cannot communicate with another operational site in the network. It is usually caused by the failure of the switching nodes and/or the links of the communicating system. The failure of a switching node includes system failure and secondary storage failure, and a link failure includes physical rupture and noise in the communication channels. Note that a communication medium failure (although it depends upon the topology and the connectivity) may not cause a total shut down of communication facilities. For example, a communication medium failure may simply cause a message loss, the receipt of a message with some errors, or the partition of a network where a subset of sites may be unable to communicate with the sites in another subset, though sites within a subset can communicate with each other.

12.4 BACKWARD AND FORWARD ERROR RECOVERY

Recall that an error is that part of the state that differs from its intended value and can lead to a system failure, and failure recovery is a process that involves restoring an erroneous state to an error-free state. There are two approaches for restoring an erroneous state to an error-free state [32]:

- If the nature of errors and damages caused by faults can be completely and accurately assessed, then it is possible to remove those errors in the process's (system's) state and enable the process (system) to move forward. This technique is known as *forward-error recovery*.
- If it is not possible to foresee the nature of faults and to remove all the errors in the process's (system's) state, then the process's (system's) state can be restored to a previous error-free state of the process (system). This technique is known as *backward-error recovery*.

Note that backward-error recovery is simpler than forward-error recovery as it is independent of the fault and the errors caused by the fault. Thus, a system can recover from an arbitrary fault by restoring to a previous state. This generality enables backward-error recovery to be provided as a general recovery mechanism to any type of system.

The major problems associated with the backward-error recovery approach are:

- Performance penalty: The overhead to restore a process (system) state to a prior state can be quite high.
- There is no guarantee that faults will not occur again when processing begins from a prior state.
- Some component of the system state may be unrecoverable. For example, cash dispensed at an automatic teller machine cannot be recovered.

The forward-error recovery technique, on the other hand, incurs less overhead because only those parts of the state that deviate from the intended value need to be corrected. However, this technique can be used only where the damages due to faults can be correctly assessed, and hence it is not a concept as general as the backward-error recovery and cannot be provided as a general mechanism for error recovery. In the forthcoming sections, we focus on backward-error recovery and several techniques to implement it in detail.

12.5 BACKWARD-ERROR RECOVERY: BASIC APPROACHES

In backward-error recovery, a process is restored to a prior state in the hope that the prior state is free of errors [32]. The points in the execution of a process to which the process can later be restored are known as *recovery points*. A recovery point is said to be restored when the current state of a process is replaced by the state of the process at the recovery point. The above concepts and the discussion that follow are also applicable at the system level. Recovery done at the process level is simply a subset of the actions necessary to recover the entire system. In a system recovery, all the user processes that were active need to be restored to their respective recovery points and data (in secondary storage) modified by the processes need to be restored to a proper state.

There are two ways to implement backward-error recovery, namely, the operation-based approach and the state-based approach [22]. These approaches are explained in the context of the following system model.

SYSTEM MODEL. The system is assumed to consist of a single machine. The machine is connected to a secondary storage system and a stable storage system (see Fig. 12.2). A storage that does not lose information in the event of system failure is referred to as a *stable* storage. Whenever a process accesses a data object stored on the secondary storage, the data object is brought into the main memory if it is not already there. If the access is a write operation, the copy of the object in the main memory is updated. The data object in the secondary storage is eventually updated when the copy of the object in the main memory is flushed to the disk by the paging scheme or when the process updating the object terminates. The stable storage is used to store the logs (defined later) and recovery points. The contents of both the stable storage and secondary storage survive system failures. However, the contents of the stable storage are much more

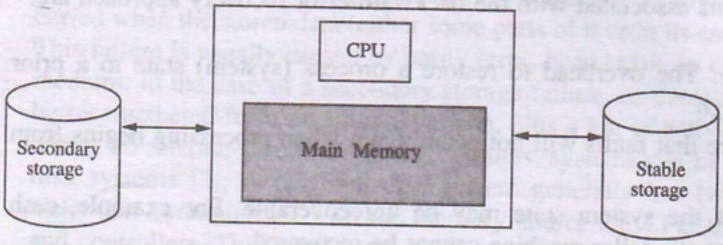

FIGURE 12.2
A system model.

secure than those of the secondary storage. It is assumed that the data on the secondary storage is archived periodically.

12.5.1 The Operation-based Approach

In the operation-based approach, all the modifications that are made to the state of a process are recorded in sufficient detail so that a previous state of the process can be restored by reversing all the changes made to the state. The record of the system activity is known as an *audit trail* or a *log* [32].

Consider a transaction based environment where transactions update a database. In such an environment, it is desirable to be able to commit or undo updates on a per-transaction basis. Commit is an action which indicates that the process or transaction updating the object has successfully completed, and therefore the changes done to the database can be made permanent. (Commit actions are explained in Chap. 13.) Note that even before a transaction commits, its updates may be recorded in the database because of the underlying paging scheme. Therefore, if a transaction does not commit, its database updates should be undone. Moreover, if a part of the database is lost due to a storage media error, it should be possible to reconstruct that part. We next describe the updating-in-place scheme proposed in [11], in which the above requirements can be satisfied.

UPDATING-IN-PLACE. Under the updating-in-place scheme, every update (write) operation to an object updates the object and results in a log to be recorded in a stable storage which has enough information to completely undo and redo the operation. The information recorded includes: (1) the name of the object, (2) the old state of the object (used for UNDO), and (3) the new state of the object (used for REDO). A recoverable update operation can be implemented as a collection of operations as follows:

- A *do* operation, which does the action (update) and writes a log record.
- An *undo* operation, which, given a log record written by a *do* operation, undoes the action performed by the *do* operation.
- A *redo* operation, which, given a log record written by a *do* operation, redoes the action specified by the *do* operation.
- An optional *display* operation, which displays the log record.

When a transaction is not committed or fails, the changes made by the transaction to the database can be undone by using *undo* operations. On the other hand, if a portion of the database is to be reconstructed, it can be reconstructed by performing *redo* operations on that previously archived portion of the database.

The major problem with the updating-in-place is that a *do* operation cannot be undone if the system crashes after an update operation but before the log record is stored. This problem is overcome by the write-ahead-log protocol [11].

THE WRITE-AHEAD-LOG PROTOCOL. In the write ahead log protocol, a recoverable update operation is implemented by the following operations:

- Update an object only after the *undo* log is recorded.
- Before committing the updates, *redo* and *undo* logs are recorded.

On restarting a system after failure (due to hardware failure or any other reason), it may be necessary to undo the changes made by the transactions that were in progress at the time of failure. Moreover, on restart, *redo* operations may have to be performed if the objects updated were still in the main memory at the time of the system failure. Therefore, both *undo* and *redo* actions should work properly, even under repetitive failures, whether updating-in-place or write-ahead-log protocol is used. Note also that writing a log record on every update operation is expensive in terms of storage requirement and CPU overhead incurred, especially if failures are rare.

12.5.2 State-based Approach

In the state-based approach for recovery, the complete state of a process is saved when a recovery point is established and recovering a process involves reinstating its saved state and resuming the execution of the process from that state [8, 32]. The process of saving state is also referred to as *checkpointing* or *taking a checkpoint*. The recovery point at which checkpointing occurs is often referred to as a *checkpoint*. The process of restoring a process to a prior-state is referred to as *rolling back* the process. Note that since rolling back a process and resuming its execution from a prior state incurs overhead and delays the completion of the process, it is desirable to rollback a process to a state as recent as possible. Therefore, it is customary to take many checkpoints over the execution of a process.

A NOTE. Readers should not construe that the state- and operation-based approaches are mutually exclusive. They can be combined together to minimize the amount of rollback in the event of a failure. Section 12.9 describes a technique that makes use of both approaches.

SHADOW PAGES. A special case of the state-based recovery approach is the technique based on shadow pages [21]. Under this technique, only a part of the system state is saved to facilitate recovery. Whenever a process wants to modify an object, the page containing the object is duplicated and is maintained on stable storage. From

that point onwards, only one of the copies undergoes all the modifications done by the process. The other unmodified copy is known as the *shadow page*. If the process fails, the modified copy is discarded to restore the database to a proper state. If the process successfully commits, then the shadow page is discarded and the modified page is made part of the database.

12.6 RECOVERY IN CONCURRENT SYSTEMS

In concurrent systems, several processes cooperate by exchanging information to accomplish a task. The information exchange can be through a shared memory in the case of shared memory machines (e.g., multiprocessor systems) or through messages in the case of a distributed system. In such systems, if one of the cooperating processes fails and resumes execution from a recovery point, then the effects it has caused at other processes due to the information it has exchanged with them after establishing the recovery point will have to be undone. To undo the effects caused by a failed process at an active process, the active process must also rollback to an earlier state. Thus, in concurrent systems, all cooperating processes need to establish recovery points. Rolling back processes in concurrent systems is more difficult than in the case of a single process. The following discussion illustrates how the rolling back of processes can cause further problems.

12.6.1 Orphan Messages and the Domino Effect

Consider the system activity illustrated in Fig. 12.3. X, Y, and Z are three processes that cooperate by exchanging information (shown by the arrows). Each symbol '[' marks a recovery point to which a process can be rolled back in the event of a failure.

If process X is to be rolled back, it can be rolled back to the recovery point x_3 without affecting any other process. Suppose that Y fails after sending message m and is rolled back to y_2. In this case, the receipt of m is recorded in x_3, but the sending of m is not recorded in y_2. Now we have a situation where X has received message

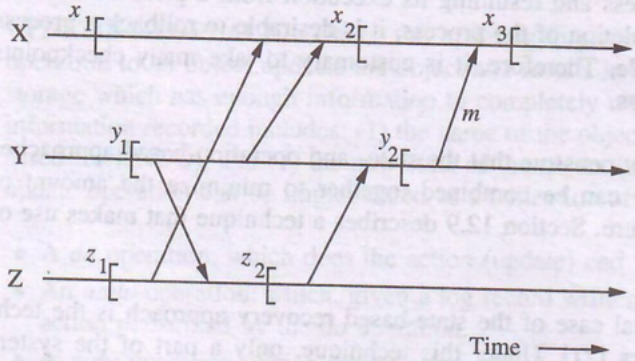

FIGURE 12.3
Domino effect.

m from Y, but Y has no record of sending it, which corresponds to an inconsistent state. Under such circumstances, m is referred to as an *orphan* message and process X must also roll back. X must roll back because Y interacted with X after establishing its recovery point y_2. When Y is rolled back to y_2, the event that is responsible for the interaction is undone. Therefore, all the effects at X caused by the interaction must also be undone. This can be achieved by rolling back X to recovery point x_2. Likewise, it can be seen that, if Z is rolled back, all three processes must roll back to their very first recovery points, namely, x_1, y_1, and z_1. This effect, where rolling back one process causes one or more other processes to roll back, is known as the *domino effect* [32], and orphan messages are the cause.

12.6.2 Lost Messages

Suppose that checkpoints x_1 and y_1 (Fig. 12.4) are chosen as the recovery points for processes X and Y, respectively. In this case, the event that sent message m is recorded in x_1, while the event of its receipt at Y is not recorded in y_1. If Y fails after receiving message m, the system is restored to state $\{x_1, y_1\}$, in which message m is *lost* as process X is past the point where it sends message m. This condition can also arise if m is lost in the communication channel and processes X and Y are in state x_1 and y_1, respectively. Both the above conditions are indistinguishable.

12.6.3 Problem of Livelocks

In rollback recovery, livelock is a situation in which a single failure can cause an infinite number of rollbacks, preventing the system from making progress [19]. A livelock situation in a distributed system is illustrated in Fig. 12.5.

Figure 12.5(a) illustrates the activity of two processes X and Y until the failure of Y. Process Y fails before receiving message n_1, sent by X. When Y rolls back to y_1, there is no record of sending message m_1, hence X must rollback to x_1. When process Y recovers, it sends out m_2 and receives n_1 (see Fig. 12.5(b)). Process X, after resuming from x_1, sends n_2 and receives m_2. However, because X rolled back, there is no record of sending n_1 and hence Y has to roll back for the second time. This forces X to rollback too, as it has received m_2, and there is no record of sending m_2 at Y. This situation can repeat indefinitely, preventing the system from making any progress.

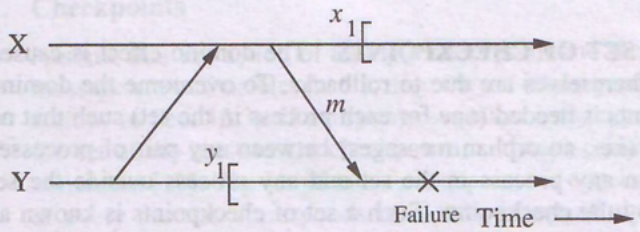

FIGURE 12.4
Message loss due to roll back recovery.

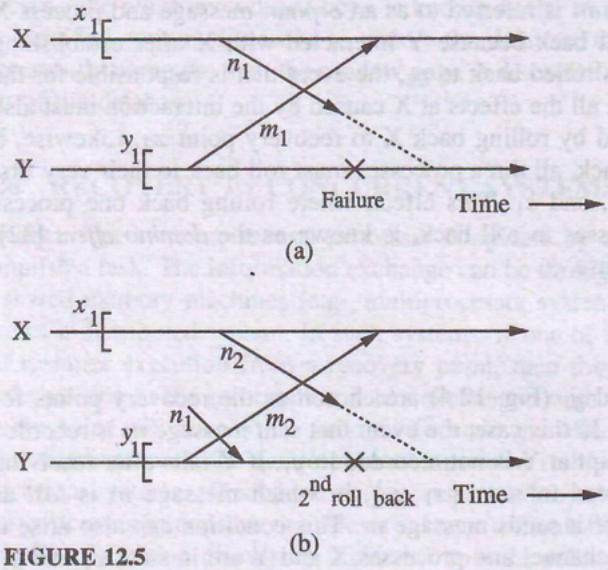

FIGURE 12.5
Livelock (adapted from [19]).

In view of these problems, operation-based or state-based recovery techniques are not adequate in locating and/or establishing usable recovery points for all the cooperating processes. There is a need for coordination among the processes, either at the time of establishing checkpoints or at the beginning of a recovery. We devote the rest of this chapter to the discussion of checkpointing and recovery in distributed systems.

12.7 CONSISTENT SET OF CHECKPOINTS

From the previous discussion, it is clear that checkpointing in distributed systems involves taking a checkpoint by all the processes (sites) or at least by a set of processes (sites) that interact with one another in performing a distributed computation. Typically, in distributed systems, all the sites save their local states, which are known as *local checkpoints,* and the process of saving local states is called *local checkpointing*. All the local checkpoints, one from each site, collectively form a *global checkpoint.*

STRONGLY CONSISTENT SET OF CHECKPOINTS. The domino effect is caused by orphan messages, which themselves are due to rollbacks. To overcome the domino effect, a set of local checkpoints is needed (one for each process in the set) such that no information flow takes place (i.e., no orphan messages) between any pair of processes in the set, as well as between any process in the set and any process outside the set during the interval spanned by the checkpoints. Such a set of checkpoints is known as a *recovery line* or a *strongly consistent set of checkpoints* [32].

In Fig. 12.6, the set $\{x_1, y_1, z_1\}$ is a *strongly* consistent set of checkpoints and the thinly dotted lines delineate the interval spanned by the checkpoints. A strongly

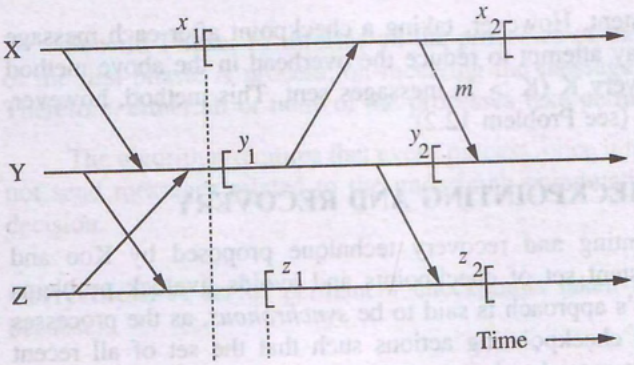

FIGURE 12.6
Consistent set of checkpoints.

consistent set of checkpoints corresponds to a strongly consistent global state (discussed in Sec. 5.6) wherein all messages have been delivered and processed, and no message is in transit. Notice that, processes X, Y, and Z can be rolled back to their respective checkpoints x_1, y_1, and z_1 and resume execution in the event of a failure. No further rollbacks due to the domino effect would be necessary as no information exchange took place in the interval spanned by the set of checkpoints. That is, no local checkpoint includes an effect whose cause would be undone due to the rollback of another process.

CONSISTENT SET OF CHECKPOINTS. Suppose that Y fails after receiving message m. If Y restarts from checkpoint y_2, message m is lost due to rollback. Note that the set $\{x_2, y_2, z_2\}$ is not a strongly consistent set of checkpoints, rather it is referred to as a *consistent set of checkpoints*. A consistent set of ckeckpoints is similar to a consistent global state (discussed in Sec. 5.6) in that it requires that each message recorded as received in a checkpoint (state) should also be recorded as sent in another checkpoint (state). Therefore, systems that do not establish a strongly consistent set of checkpoints have to deal with lost messages during roll back recovery. While the systems which establish strongly consistent set of checkpoints do not have to deal with lost messages during roll back recovery, they experience delays during the checkpointing process as processes cannot exchange messages while checkpointing is in progress.

12.7.1 A Simple Method for Taking a Consistent Set of Checkpoints

Assume that the action of taking a checkpoint and the action of sending or receiving a message are indivisible; that is, they are not interrupted by any other events. (See atomic actions in Sec. 13.3). If every process takes a checkpoint after sending every message, the set of the most recent checkpoints is always consistent. However, it is not strongly consistent [19]. The set of latest checkpoints is consistent because the latest checkpoint at every process corresponds to a state where all the messages recorded as received in it have already been recorded elsewhere as sent. Therefore, rolling back a process to its latest checkpoint would not result in any orphan messages, which would cause

the system state to be inconsistent. However, taking a checkpoint after each message is sent is expensive, so one may attempt to reduce the overhead in the above method by taking a checkpoint after every K (K > 1) messages sent. This method, however, suffers from the domino effect (see Problem 12.2).

12.8 SYNCHRONOUS CHECKPOINTING AND RECOVERY

We now describe a checkpointing and recovery technique proposed by Koo and Toueg [19] that takes a consistent set of checkpoints and avoids livelock problems during recovery. The algorithm's approach is said to be *synchronous*, as the processes involved coordinate their local checkpointing actions such that the set of all recent checkpoints in the system is guaranteed to be consistent [17].

12.8.1 The Checkpoint Algorithm

The checkpoint algorithm assumes the following characteristics for the distributed system:

- Processes communicate by exchanging messages through communication channels.
- Channels are FIFO in nature. End-to-end protocols (such as sliding window protocols [41]) are assumed to cope with message loss due to rollback recovery (Fig. 12.4) and communication failure. (Another way to handle message loss is to have processes log messages in stable storage before sending them. A process encountering message loss due to a rollback can request the retransmission of the message. This scheme, however, requires that every process record the identity of the last message it has received from each process on stable storage.)
- Communication failures do not partition the network.

The checkpoint algorithm takes two kinds of checkpoints on stable storage, permanent and tentative. A permanent checkpoint is a local checkpoint at a process and is a part of a consistent global checkpoint. A tentative checkpoint is a temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm. Processes roll back only to their permanent checkpoint.

The checkpoint algorithm assumes that a single process invokes the algorithm, as opposed to several processes concurrently invoking the algorithm to take permanent checkpoints. Furthermore, the algorithm assures that no site in the distributed system fails during the execution of the algorithm.

The algorithm has two phases.

First Phase. An initiating process P_i takes a tentative checkpoint and requests all the processes to take tentative checkpoints. Each process informs P_i whether it succeeded in taking a tentative checkpoint. A process says "no" to a request if it fails to take a checkpoint, which could be due to several reasons, depending upon the underlying application. If P_i learns that all the processes have successfully taken tentative checkpoints, P_i decides that all tentative checkpoints should be made permanent; otherwise, P_i decides that all the tentative checkpoints should be discarded.

Second Phase. P_i informs all the processes of the decision it reached at the end of the first phase. A process, on receiving the message from P_i, will act accordingly. Therefore, either all or none of the processes take permanent checkpoints.

The algorithm requires that every process, once it has taken a tentative checkpoint, not send messages related to the underlying computation until it is informed of P_i's decision.

Correctness. A set of permanent checkpoints taken by this algorithm is consistent because:

- Either all or none of the processes take permanent checkpoints.
- A set of checkpoints will be inconsistent if there is a record of a message received but not of the event sending it. This will not happen as no process sends messages after taking a tentative checkpoint until the receipt of the initiating process's decision, by which time all processes would have taken checkpoints.

OPTIMIZATION. While the above protocol takes a consistent set of checkpoints, it may cause a process to take a checkpoint even when it is not necessary (note that taking a checkpoint is an expensive operation).

For example, consider the system activity shown in Fig. 12.7. The set $\{x_1, y_1, z_1\}$ is a consistent set of checkpoints. Suppose process X decides to initiate the checkpointing algorithm after receiving message m. It takes a tentative checkpoint x_2 and sends "take tentative checkpoint messages" to processes Y and Z, causing Y and Z to take checkpoints y_2 and z_2, respectively. Now $\{x_2, y_2, z_2\}$ forms a consistent set of checkpoints. Note, however, that $\{x_2, y_2, z_1\}$ also forms a consistent set of checkpoints. (The checkpoint algorithm uses the weaker definition of consistency which requires that every message recorded as "received" in a checkpoint should also be recorded as "sent" in another checkpoint, and not vice versa [19].) In our example, process Y should take

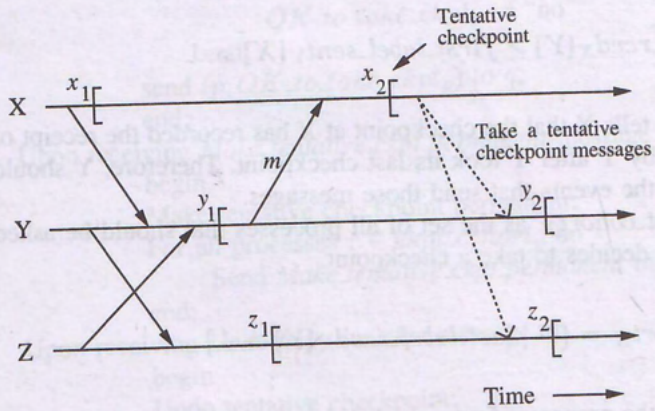

FIGURE 12.7
Checkpoints taken unnecessarily.

a checkpoint since x_2 records the receipt of message m, and y_1 does not record the sending of message m. However, there is no need for process Z to take checkpoint x_2 because Z has not sent any message since its last checkpoint. A process can decide whether it is necessary to take a checkpoint or not with the help of a labeling scheme described below.

Messages that are sent by the checkpointing or rollback-recovery algorithms (discussed later) are referred to as *control* messages. Messages that are sent as a part of the underlying computation are referred to as *messages*. Every outgoing message m has a field for a label, denoted by $m.l$. Each process uses monotonically increasing labels in its outgoing messages. The following terminology will be used in describing the algorithm:

$$\perp = \text{smallest label}$$
$$\top = \text{largest label.}$$

For any two processes X and Y, let m be the last message that X received from Y after X has taken its last permanent or tentative checkpoint. Then

$$last_label_rcvd_X[Y] = \begin{cases} m.l \text{ if } m \text{ exists} \\ \perp \quad \text{otherwise} \end{cases}$$

Let m be the first message that X sent to Y after X took its last permanent or tentative checkpoint. Then

$$first_label_sent_X[Y] = \begin{cases} m.l \text{ if } m \text{ exists} \\ \perp \quad \text{otherwise} \end{cases}$$

Whenever X requests Y to take a tentative checkpoint, X sends $last_label_rcvd_X[Y]$ along with its request; Y takes a tentative checkpoint only if

$$last_label_rcvd_X[Y] \geq first_label_sent_Y[X] > \perp$$

The above condition simply tells Y that the checkpoint at X has recorded the receipt of one or more messages sent by Y after Y took its last checkpoint. Therefore, Y should take a checkpoint to record the events that send those messages.

Finally, we define $ckpt_cohort_X$ as the set of all processes that should be asked to take checkpoints when X decides to take a checkpoint.

$$ckpt_cohort_X = \{Y \mid last_label_rcvd_X[Y] > \perp\}$$

This set simply indicates all the processes from which X has received messages after it has taken its last checkpoint. If X takes a checkpoint, then those processes should also take checkpoints to record the sending of those messages.

OUTLINE OF THE ALGORITHM

Initial state at all processes p:

for all processes q do $first_label_sent_p[q] := \perp$;

$$OK_to_take_ckpt_p = \begin{cases} \text{"yes" if } p \text{ is willing to take a checkpoint} \\ \text{"no" otherwise} \end{cases}$$

At initiator process P_i:

for all processes $p \in ckpt_cohort_{P_i}$ do
  send $Take_a_tentative_ckpt(P_i, last_label_rcvd_{P_i}[p])$ message;
if all processes replied "yes" then
  for all processes $p \in ckpt_cohort_{P_i}$ do
    send $Make_tentative_ckpt_permanent$;
else
  for all processes $p \in ckpt_cohort_{P_i}$ do
    send $Undo_tentative_ckpt$.

At all processes p:

Upon receiving $Take_a_tentative_ckpt(q, last_label_rcvd_q[p])$ message from q do
  begin
    if $OK_to_take_ckpt_p$ = "yes" AND
    $last_label_rcvd_q[p] \geq first_label_sent_p[q] > \perp$ then
      begin
      take a tentative checkpoint;
      for all processes $r \in ckpt_cohort_p$ do
        send $Take_a_tentative_ckpt(p, last_label_rcvd_p[r])$ message;
      if all processes $r \in ckpt_cohort_p$ replied "yes" then
        $OK_to_take_ckpt_p$:= "yes"
      else
        $OK_to_take_ckpt_p$:= "no"
      end;
    send $(p, OK_to_take_ckpt_p)$ to q;
  end;

Upon receiving *Make_tentative_ckpt_permanent* message do
  begin
  Make tentative checkpoint permanent;
  For all processes $r \in ckpt_cohort_p$ do
    Send *Make_tentative_ckpt_permanent* message;
  end;

Upon receiving *Undo_tentative_ckpt* message do
  begin
  Undo tentative checkpoint;
  For all processes $r \in ckpt_cohort_p$ do
    Send *Undo_tentative_ckpt* message;
  end;

12.8.2 The Rollback Recovery Algorithm

The rollback recovery algorithm assumes that a single process invokes the algorithm, as opposed to several processes concurrently invoking it to rollback and recover [19]. It also assumes that the checkpoint and the rollback recovery algorithms are not concurrently invoked. The rollback recovery algorithm has two phases.

First Phase. An initiating process P_i checks to see if all the processes are willing to restart from their previous checkpoints. A process may reply "no" to a restart request if it is already participating in a checkpointing or a recovering process initiated by some other process. If P_i learns that all the processes are willing to restart from their previous checkpoints, P_i decides that all the processes should restart; otherwise, P_i decides that all the processes should continue with their normal activities. (P_i may attempt a recovery at a later time.)

Second Phase. P_i propagates its decision to all the processes. On receiving P_i's decision, a process will act accordingly.

The recovery algorithm requires that every process not send messages related to the underlying computation while it is waiting for P_i's decision.

Correctness. All cooperating processes restart from an appropriate state because:

- All processes either restart from their previous checkpoints or continue with their normal activities.
- If processes decide to restart, then they resume execution in a consistent state, as the checkpoint algorithm (Sec. 12.8.1) takes a consistent set of checkpoints.

OPTIMIZATION. While the above protocol causes all the processes to restart from a consistent set of checkpoints (taken by the checkpointing algorithm), it causes all the processes to roll back irrespective of whether a process needs to roll back or not. For example, consider the process activity shown in Fig. 12.8. The above protocol, in the event of failure of process X, would require processes X, Y, and Z to restart from checkpoints x_2, y_2, and z_2, respectively. Note, however, that process Z need not have rolled back as there was no interaction between Z and the other two processes.

To minimize the number of process rollbacks, the rollback recovery algorithm uses the labeling scheme explained in Sec. 12.8.1. In addition to the terminology previously introduced, the following terminology is used in describing the rollback recovery algorithm: For any two processes X and Y, let m be the last message that X sent to Y before X takes its latest permanent checkpoint. Then

$$last_label_sent_X[Y] = \begin{cases} m.l & \text{if } m \text{ exists} \\ \top & \text{otherwise} \end{cases}$$

When X requests Y to restart from the permanent checkpoint, it sends $last_label_sent_X[Y]$ along with its request. Y will restart from its permanent checkpoint only if

$$last_label_rcvd_Y[X] > last_label_sent_X[Y]$$

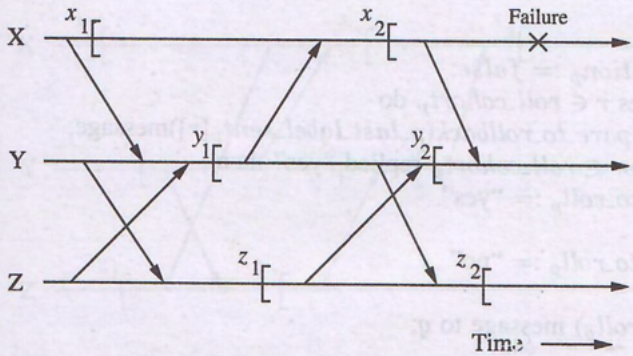

FIGURE 12.8
Unnecessary rollback.

When this condition holds, it indicates that X is rolling back to a state where the sending of one or more messages from X to Y is being undone.
We also define:

$$roll_cohort_X = \{Y \mid X \textit{ can send messages to } Y\}$$

OUTLINE OF THE ALGORITHM

Initial state at all processes p:

$resume_execution_p$:= true;
for all processes q, do
$last_label_rcvd_p[q] := \top$;

$$willing_to_roll_p = \begin{cases} \text{"yes" if } p \text{ is willing to roll back} \\ \text{"no" otherwise} \end{cases}$$

At initiator process P_i:

for all processes $p \in roll_cohort_{P_i}$ do
send $Prepare_to_rollback(P_i, last_label_sent_{P_i}[p])$ message;
if all processes replied "yes" then
for all processes $p \in roll_cohort_{P_i}$ do
send *Roll_back* message;
else
for all processes $p \in roll_cohort_{P_i}$ do
send *Donot_roll_back* message;

At all processes p:

Upon receiving $Prepare_to_rollback(q, last_label_sent_q[p])$ message from q do
begin
if $willing_to_roll_p$ AND $last_label_rcvd_p[q] > last_label_sent_q[p]$ AND $(resume_execution_p)$

```
        then
            begin
            resume_execution_p := false;
            for all processes r ∈ roll_cohort_p do
                send Prepare_to_rollback(p, last_label_sent_p[r])message;
            if all processes r ∈ roll_cohort_p replied "yes" then
                willing_to_roll_p := "yes"
            else
                willing_to_roll_p := "no"
            end;
        Send(p, willing_to_roll_p) message to q;
        end;
    Upon receiving Roll_back message AND if resume_execution_p = false do
        begin
        restart from p's permanent checkpoint;
        for all processes r ∈ roll_cohort_p do
            send Roll_back message;
        end;
    Upon receiving Donot_roll_back message do
        begin
        resume execution;
        for all processes r ∈ roll_cohort_p do
            send Donot_roll_back message;
        end;
```

12.9 ASYNCHRONOUS CHECKPOINTING AND RECOVERY

While synchronous checkpointing simplifies recovery (because a consistent set of checkpoints is readily available), it has the following disadvantages [17]:

1. Additional messages are exchanged by the checkpoint algorithm when it takes each checkpoint.
2. Synchronization delays are introduced during normal operations. (Note that in the synchronous checkpointing algorithm described previously, no computational messages can be sent while the checkpointing algorithm is in progress.)
3. If failures rarely occur between successive checkpoints, then the synchronous approach places unnecessary burden on the system in the form of additional messages, delays, and processing overhead.

Under the asynchronous approach, checkpoints at each processor (or process) are taken independently without any synchronization among the processors [17]. Because of the absence of synchronization, there is no guarantee that a set of local checkpoints taken will be a consistent set of checkpoints. Thus, a recovery algorithm has to search