CHAPTER 13

FAULT TOLERANCE

13.1 INTRODUCTION

In the previous chapter, several techniques to recover from failures were discussed. However, the disruptions caused during failures can be especially severe in many cases (for example: on-line transaction processing, process control, and computer based communication user communities, etc.) [14]. To avoid disruptions due to failures and to improve availability, systems are designed to be fault-tolerant.

A system can be designed to be fault-tolerant in two ways [14]. A system may *mask* failures or a system may exhibit a *well defined failure behavior* in the event of failure. When a system is designed to mask failures, it continues to perform its specified function in the event of a failure. A system designed for well defined behavior may or may not perform the specified function in the event of a failure, however, it can facilitate actions suitable for recovery. An example of well defined behavior during a failure is: the changes made to a database by a transaction are made visible to other transactions only if the transaction successfully commits; if the transaction fails, the changes made to the database by the failed transaction are not made visible to the other transactions, thus not affecting those transactions.

One key approach used to tolerate failures is *redundancy*. In this approach, a system may employ a multiple number of processes, a multiple number of hardware components, multiple copies of data, etc., each with independent failure modes (i.e., failure of one component does not affect the operation of other components).

In this chapter, we discuss widely used techniques, such as commit protocols and voting protocols, used in the design of fault-tolerant systems. Commit protocols

implement well defined behavior in the event of failure, such as the one described in the above example. Voting protocols, on the other hand, mask failures in a system. To implement a fault-tolerant distributed system, processes in the system should be able to tolerate system failures and communicate reliably. We describe two techniques that have been used to implement processes that are resilient to system failures. In addition, we describe a technique to send messages reliably among processes. Finally, we close this chapter by presenting a case study of a fault-tolerant system.

13.2 ISSUES

Since a fault-tolerant system must behave in a specified manner in the event of a failure, it is important to study the implications of certain types of failures.

PROCESS DEATHS. When a process dies, it is important that the resources allocated to that process are recouped, otherwise they may be permanently lost. Many distributed systems are structured along the client-server model in which a client requests a service by sending a message to a server. If the server process fails, it is necessary that the client machine be informed so that the client process, waiting for a reply can be unblocked to take suitable action. Likewise, if a client process dies after sending a request to a server, it is imperative that the server be informed that the client process no longer exists. This will facilitate the server in reclaiming any resources it has allocated to the client process.

MACHINE FAILURE. In the case of machine failure, all the processes running at the machine will die. As far as the behavior of a client process or a server process is concerned, there is not much difference in their behavior in the event of a machine failure or a process death. The only difference lies in how the failure is detected. In the case of a process death, other processes including the kernel remain active. Hence, a message stating that the process has died can be sent to an inquiring process. On the other hand, an absence of any kind of message indicates either process death or a failure due to machine failure.

NETWORK FAILURE. A communication link failure can partition a network into subnets, making it impossible for a machine to communicate with another machine in a different subnet. A process cannot really tell the difference between a machine and a communication link failure, unless the underlying communication network (such as a slotted ring network) can recognize a machine failure. If the communication network cannot recognize machine failures and thus cannot return a suitable error code (such as Ethernet), a fault-tolerant design will have to assume that a machine may be operating and processes on that machine are active.

13.3 ATOMIC ACTIONS AND COMMITTING

Typically, system activity is governed by the sequence of primitive or atomic operations it is executing. Usually, a machine level instruction, which is indivisible, instantaneous,

and cannot be interrupted (unless the system fails) corresponds to an atomic operation. However, it is desirable to be able to group such instructions that accomplish a certain task and make the group an atomic operation.

For example, suppose two processes P_1 and P_2 share a memory location X and both modify X as shown in Fig. 13.1. Suppose P_1 succeeds in locking X before P_2, then P_1 updates X and releases the lock, making it possible for P_2 to access X. If P_1 fails after P_2 has seen the changes made to X by P_1, then P_2 will also have to be aborted or rolled back. Thus, what is necessary is that P_2 should not be able to interact with P_1 through X until it can do so safely. In other words, P_1 should be atomic. Its effect on X should not be visible to P_2 or any other process until P_1 is guaranteed to finish. In essence, the effect of P_1 on the system (even though it executes concurrently with P_2) should look like an undivided and uninterrupted operation.

Atomic actions extend the concept of atomicity from one machine instruction level to a sequence of instructions or a group of processes that are themselves to be executed atomically. Atomic actions are the basic building blocks in constructing fault-tolerant operations. They provide a means to a system designer to specify the process interactions that are to be prevented to maintain the integrity of the system. Atomic actions have the following characteristics [29, 39].

- An action is atomic if the process performing it is not aware of the existence of any other active processes, and no other process is aware of the activity of the process during the time the process performs the action.
- An action is atomic if the process performing it does not communicate with other processes while the action is being performed.
- An action is atomic if the process performing it can detect no state changes except those performed by itself, and if it does not reveal its state changes until the action is complete.
- Actions are atomic if they can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

A transaction groups a sequence of actions (for example, on a database) and the group is treated as an atomic action to maintain the consistency of a database. (The concept of a transaction is discussed in Sec. 19.2.1.) At some point during its

Process P_1	**Process** P_2
–	–
–	–
Lock(X);	Lock(X);
X := X + Z;	X := X + Y;
Unlock(X);	Unlock(X);
–	–
–	–
failure	

FIGURE 13.1
Process interaction.

execution, the transaction decides whether to commit or abort its actions. A *commit* is an unconditional guarantee (even in the case of multiple failures) that the transaction will be completed. In other words, the effects of its actions on the database will be permanent. An *abort* is an unconditional guarantee to back out of the transaction, and none of the effects of its actions will persist [44].

A transaction may abort due to any of the following events: deadlocks, timeouts, protection violation, wrong input provided by user, or consistency violations (which can happen if an optimistic concurrency control technique is employed). To facilitate backing out of an aborting transaction, the write-ahead-log protocol (discussed in Sec. 12.5.1) or shadow pages (discussed in Sec. 12.5.1) can be employed.

In distributed systems, several processes may coordinate to perform a task. Their actions may have to be atomic with respect to other processes. For example, transaction may spawn many processes that are executed at different sites. As another example, in distributed database systems, a transaction must be processed at every site or at none of the sites to maintain the integrity of the database. This is referred to as *global atomicity*. The protocols that enforce global atomicity are referred to as *commit protocols*. Given that each site has a recovery strategy (e.g., the write-ahead-log protocol or the shadow page protocol) at the local level, commit protocols ensure that all the sites either commit or abort the transaction unanimously, even in the presence of multiple and repetitive failures [44]. Note that commit protocols fall into the second class of fault-tolerant design techniques in that they help the system behave in a certain way in the presence of failures. We next present several commit protocols.

13.4 COMMIT PROTOCOLS

The following situation illustrates the difficulties that arise in the design of commit protocols [20].

THE GENERALS PARADOX. There are two generals of the same army who have encamped a short distance apart. Their objective is to capture a hill, which is possible only if they attack simultaneously. If only one general attacks, he will be defeated. The two generals can communicate only by sending messengers. There is a chance that these messengers might lose their way or be captured by the enemy. The challenge is to use a protocol that allows the generals to agree on a time to attack, even though some messengers do not get through.

A simple proof shows that there exists no protocol which sends the messengers a fixed number of times to solve the above problem. Let P be the shortest protocol. Suppose the last messenger in P does not make it to the destination. Then either the message carried by the messenger is useless or one of the generals does not get the needed message. Since P is the minimal length protocol by our assumption, the message that was lost was not a useless message and hence one of the generals will not attack. This contradiction proves that there exists no such protocol P of fixed length.

The situation faced by the generals is very similar to the situation that arises in the commit protocols. The goal of commit protocols is to have all the sites (generals) agree either to commit (attack) or to abort (do not attack) a transaction. By relaxing the

requirement that the number of messages employed by a commit protocol be bounded by a fixed number of messages, a commit protocol can be designed. We next describe a famous protocol by Gray [20], which has been referred to as the two-phase commit protocol.

13.4.1 The Two-Phase Commit Protocol

This protocol assumes that one of the cooperating processes acts as a coordinator. Other processes are referred to as cohorts. (Cohorts are assumed to be executing at different sites.) This protocol assumes that a stable storage is available at each site and the write-ahead log protocol is active. At the beginning of the transaction, the coordinator sends a start transaction message to every cohort.

Phase I. *At the coordinator:*

1. The coordinator sends a COMMIT-REQUEST message to every cohort requesting the cohorts to commit.
2. The coordinator waits for replies from all the cohorts.

At cohorts:

1. On receiving the COMMIT-REQUEST message, a cohort takes the following actions. If the transaction executing at the cohort is successful, it writes UNDO and REDO log on the stable storage and sends an AGREED message to the coordinator. Otherwise, it sends an ABORT message to the coordinator.

Phase II. *At the coordinator:*

1. If all the cohorts reply AGREED and the coordinator also agrees, then the coordinator writes a COMMIT record into the log. Then it sends a COMMIT message to all the cohorts. Otherwise, the coordinator sends an ABORT message to all the cohorts.
2. The coordinator then waits for acknowledgments from each cohort.
3. If an acknowledgment is not received from any cohort within a timeout period, the coordinator resends the commit/abort message to that cohort.
4. If all the acknowledgments are received, the coordinator writes a COMPLETE record to the log (to indicate the completion of the transaction).

At cohorts:

1. On receiving a COMMIT message, a cohort releases all the resources and locks held by it for executing the transaction, and sends an acknowledgment.
2. On receiving an ABORT message, a cohort undoes the transaction using the UNDO log record, releases all the resources and locks held by it for performing the transaction, and sends an acknowledgment.

When there are no failures or message losses, it is easy to see that all sites will commit only when all the participants (including the coordinator) agree to commit. In the case of lost messages (sent from either cohorts or the coordinator), the coordinator simply resends messages after the timeout. Now we shall attempt to show that this protocol results in all participants either committing or aborting, even in the case of site failures.

SITE FAILURES. For site failures, we look at the following cases:

- Suppose the coordinator crashes before having written the COMMIT record. On recovery, the coordinator broadcasts an ABORT message to all the cohorts. All the cohorts who had agreed to commit will simply undo the transaction using the UNDO log and abort. Other cohorts will simply abort the transaction. Note that all the cohorts are blocked until they receive an ABORT message.
- Suppose the coordinator crashes after writing the COMMIT record but before writing the COMPLETE record. On recovery, the coordinator broadcasts a COMMIT message to all the cohorts and waits for acknowledgments. In this case also the cohorts are blocked until they receive a COMMIT message.
- Suppose the coordinator crashes after writing the COMPLETE record. On recovery, there is nothing to be done for the transaction.
- If a cohort crashes in Phase I, the coordinator can abort the transaction because it did not receive a reply from the crashed cohort.
- Suppose a cohort crashes in Phase II, that is, after writing its UNDO and REDO log. On recovery, the cohort will check with the coordinator whether to abort (i.e., perform an undo operation) or to commit the transaction. Note that committing may require a redo operation because the cohort may have failed before updating the database.

While the two-phase commit protocol guarantees global atomicity, its biggest drawback is that it is a blocking protocol. Whenever the coordinator fails, cohort sites will have to wait for its recovery (see Problem 13.1). This is undesirable as these sites may be holding locks on the resources. (Note that transactions lock the resources to maintain the integrity of resources. See Chap. 20.) In the event of message loss, the two-phase protocol will result in the sending of more messages. We next discuss nonblocking commit protocols that do not block in the event of site failures.

13.5 NONBLOCKING COMMIT PROTOCOLS

If transactions must be resilient† to site failures, the commit protocols must not block in the event of site failures. To ensure that commit protocols are nonblocking in the event

†Progress despite failures.

of site failures, operational sites should agree on the outcome of the transaction (while guaranteeing global atomicity) by examining their local states. In addition, the failed sites, upon recovery must all reach the same conclusion regarding the outcome (abort or commit) of the transaction. This decision must be consistent with the final outcome at the sites that were operational. If the recovering sites can decide the final outcome of the transaction based solely on their local state (without contacting the sites that were operational), the recovery is referred to as *independent recovery* [44]. Skeen [43, 44] proposed nonblocking commit protocols that tolerate site failures. Before describing a nonblocking protocol, it is first necessary to discuss the conditions that cause a commit protocol to block and then discuss how a failed site can recover to an appropriate state.

ASSUMPTIONS. The communication network is assumed to have the following characteristics:

- The network is reliable and point-to-point communication is possible between any two operational sites.
- The network can detect the failure of a site (for example by a timeout) and report it to the site trying to communicate with the failed site.

DEFINITIONS

Synchronous protocols. A protocol is said to be *synchronous* within one state transition if one site never leads another site by more than one state transition during the execution of the protocol. In other words, $\forall i, j, \mid t_i - t_j \mid \leq 1$, where $1 \leq i, j \leq n, n$ is the total number of sites, and t_k is the total number of state transitions that have occurred thus far at site k. A state transition (change in the state) occurs in a process participating in the two-phase commit protocol whenever it receives and/or sends messages (see Fig. 13.2). With the help of a finite state automaton (FSA), we will see that the two-phase commit protocol satisfies the above definition (see Fig. 13.2).

Whenever the coordinator is in state q, all the cohorts are either in state q or a. When the coordinator is in state w, a cohort can either be in state q, w, or a, which is at most one state transition behind or ahead of the coordinator's state in the FSA. When the coordinator is in state a/c, a cohort is in state w or a/c depending on whether it has received a message (Abort/Commit) from the coordinator.

Likewise, whenever a cohort is in state q: some cohorts may be in state w/q if they have or have not received the Commit_Request message yet; and some cohorts may be in state a depending on whether a cohort has received an Abort message or not. Whenever a cohort is in state a/c, other cohorts may be in state a or c, depending on whether they have received an Abort or Commit message, respectively; otherwise, they are in state w. Note that a site is never in state c when another site is in state q, which means that a site never leads another site by two or more state transitions.

Concurrency set. Let s_i denote the state of site i. The set of all the states of every site that may be concurrent with it is known as the concurrency set of s_i (denoted by $C(s_i)$). For example, consider a system having two sites. If site 2's state is w_2, then

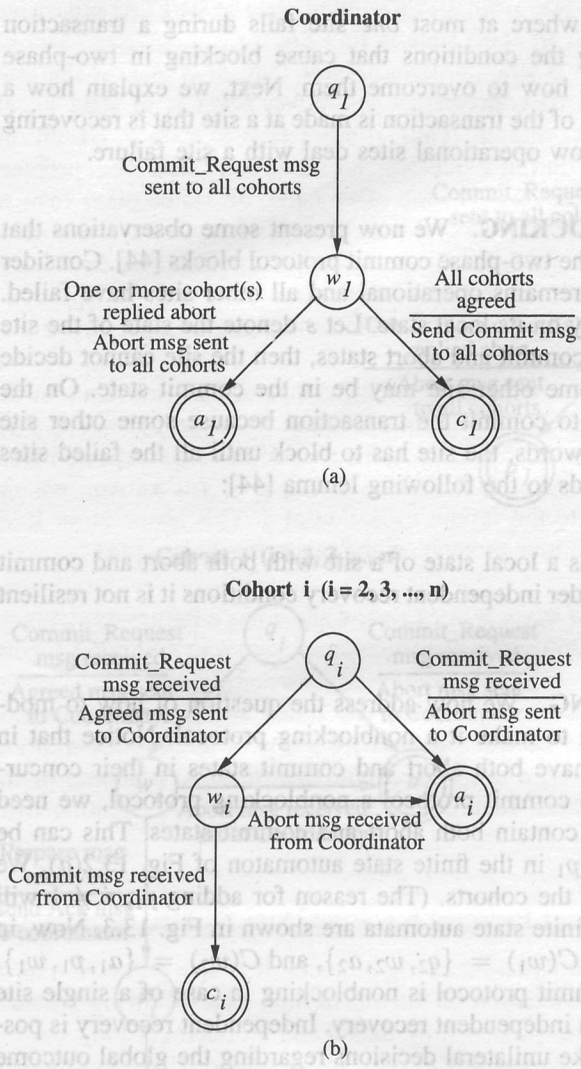

FIGURE 13.2
Finite state automata illustrating the 2-phase commit protocol (adapted from [43]).

$C(w_2) = \{c_1, a_1, w_1\}$. Likewise, $C(q_2) = \{q_1, w_1\}$. Note that, $a_1, c_1 \notin C(q_2)$ because the two-phase commit protocol is synchronous within one state transition.

Sender set. Let s be an arbitrary state of a site, and let M be the set of all messages that are received in state s. The sender set for s, denoted by $S(s)$, is

$$\{i \mid \text{site } i \text{ sends } m \text{ and } m \in M\}$$

13.5.1 Basic Idea

We first consider the simple case where at most *one* site fails during a transaction execution. We begin by describing the conditions that cause blocking in two-phase commit protocols. We then discuss how to overcome them. Next, we explain how a decision regarding the final outcome of the transaction is made at a site that is recovering after failure. Finally, we describe how operational sites deal with a site failure.

CONDITIONS THAT CAUSE BLOCKING. We now present some observations that lead to the conditions under which the two-phase commit protocol blocks [44]. Consider a simple case where only one site remains operational and all other sites have failed. This site has to proceed based solely on its local state. Let s denote the state of the site at this point. If $C(s)$ contains both commit and abort states, then the site cannot decide to abort the transaction because some other site may be in the commit state. On the other hand, the site cannot decide to commit the transaction because some other site may be in the abort state. In other words, the site has to block until all the failed sites recover. The above observation leads to the following lemma [44]:

Lemma 13.1. If a protocol contains a local state of a site with both abort and commit states in its concurrency set, then under independent recovery conditions it is not resilient to an arbitrary single failure.

HOW TO ELIMINATE BLOCKING. We now address the question of how to modify the two-phase commit protocol to make it a nonblocking protocol. Notice that in Fig. 13.2, only states w_i $(i \neq 1)$ have both abort and commit states in their concurrency sets. To make the two-phase commit protocol a nonblocking protocol, we need to make sure that $C(w_i)$ does not contain both abort and commit states. This can be done by introducing a buffer state p_1 in the finite state automaton of Fig. 13.2(a). We also introduce a buffer state p_i for the cohorts. (The reason for adding p_i, $i \neq 1$ will become clear later.) The resulting finite state automata are shown in Fig. 13.3. Now, in a system containing only two sites, $C(w_1) = \{q_2, w_2, a_2\}$, and $C(w_2) = \{a_1, p_1, w_1\}$.

This extended two-phase commit protocol is nonblocking in case of a single site failure and a failed site can perform independent recovery. Independent recovery is possible mainly because a site can make unilateral decisions regarding the global outcome of a transaction. Also, when a site fails, other sites can make decisions regarding the global outcome of the transaction based on their local states.

FAILURE TRANSITIONS. In order to perform independent recovery at a failed site, the failed site should be able to reach a final decision based solely on its local state. The decision making process is modeled in the FSA using *failure transitions*. A failure transition occurs at a failed site at the instant it fails (or immediately after it recovers from the failure). The local state resulting due to the state change caused by the failure transition will initially be occupied by the site upon recovery. The failure transitions are performed according to the following rule [44].

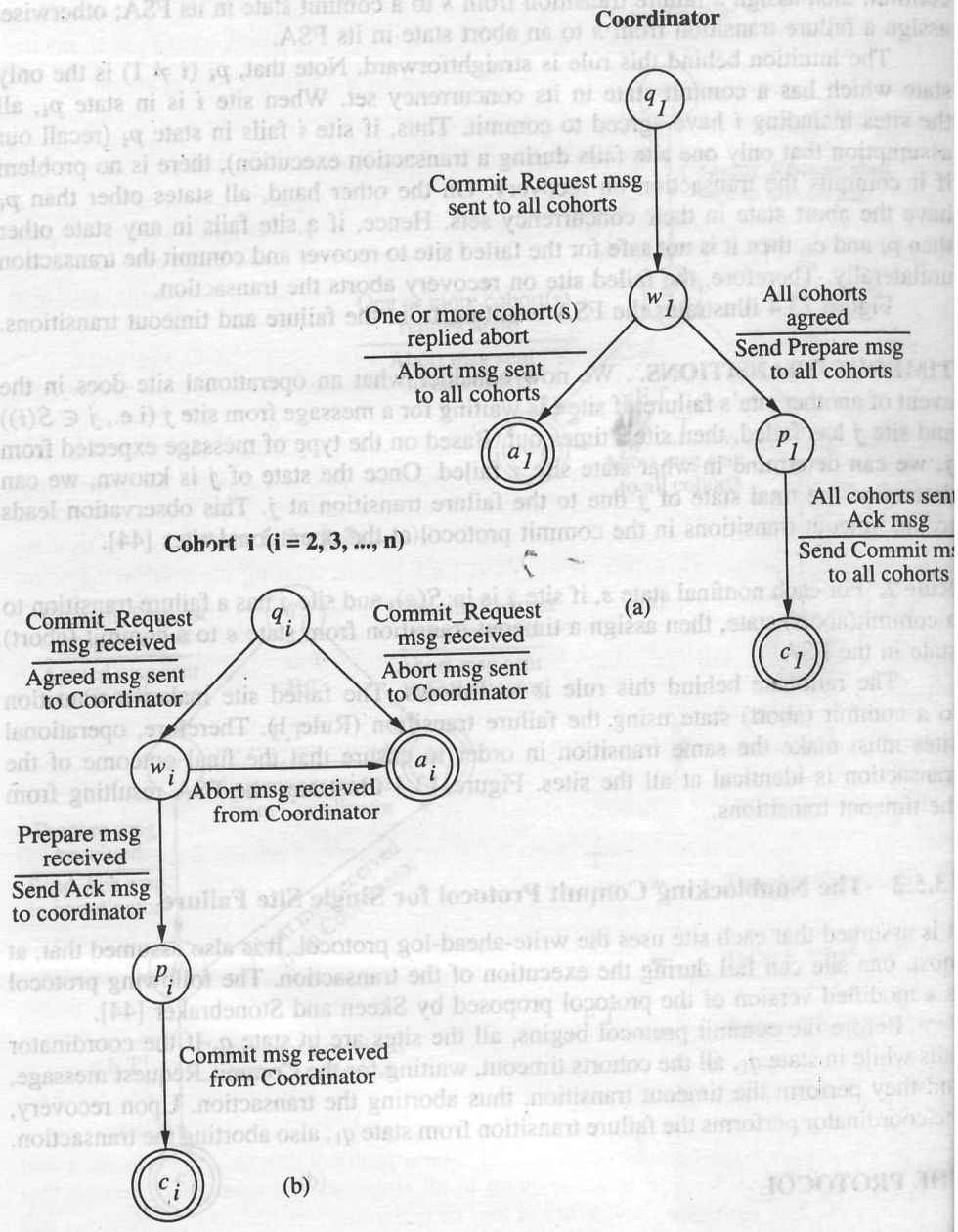

FIGURE 13.3
Finite state automata illustrating 3-phase commit protocol (adapted from Skeen [44]).

Rule 1. For every nonfinal state s (i.e., q_i, w_i, p_i) in the protocol: if $C(s)$ contains a commit, then assign a failure transition from s to a commit state in its FSA; otherwise, assign a failure transition from s to an abort state in its FSA.

The intuition behind this rule is straightforward. Note that, p_i ($i \neq 1$) is the only state which has a commit state in its concurrency set. When site i is in state p_i, all the sites including i have agreed to commit. Thus, if site i fails in state p_i (recall our assumption that only one site fails during a transaction execution), there is no problem if it commits the transaction on recovery. On the other hand, all states other than p_i have the abort state in their concurrency sets. Hence, if a site fails in any state other than p_i and c_i, then it is not safe for the failed site to recover and commit the transaction unilaterally. Therefore, the failed site on recovery aborts the transaction.

Figure 13.4 illustrates the FSA resulting from the failure and timeout transitions.

TIMEOUT TRANSITIONS. We now consider what an operational site does in the event of another site's failure. If site i is waiting for a message from site j (i.e., $j \in S(i)$) and site j has failed, then site i times out. Based on the type of message expected from j, we can determine in what state site j failed. Once the state of j is known, we can determine the final state of j due to the failure transition at j. This observation leads to the timeout transitions in the commit protocol at the operational sites [44].

Rule 2. For each nonfinal state s, if site j is in $S(s)$, and site j has a failure transition to a commit(abort) state, then assign a timeout transition from state s to a commit (abort) state in the FSA.

The rationale behind this rule is as follows. The failed site makes a transition to a commit (abort) state using the failure transition (Rule 1). Therefore, operational sites must make the same transition in order to ensure that the final outcome of the transaction is identical at all the sites. Figure 13.4 illustrates the FSA resulting from the timeout transitions.

13.5.2 The Nonblocking Commit Protocol for Single Site Failure

It is assumed that each site uses the write-ahead-log protocol. It is also assumed that, at most, one site can fail during the execution of the transaction. The following protocol is a modified version of the protocol proposed by Skeen and Stonebraker [44].

Before the commit protocol begins, all the sites are in state q. If the coordinator fails while in state q_1, all the cohorts timeout, waiting for the Commit_Request message, and they perform the timeout transition, thus aborting the transaction. Upon recovery, the coordinator performs the failure transition from state q_1, also aborting the transaction.

THE PROTOCOL

Phase I. The first phase of the nonblocking protocol is identical to that of the two-phase commit protocol (see Sec. 13.4.1) except in the event of a site's failure. During the first phase, the coordinator is in state w_1, and each cohort is either in state a (in which case the site has already sent an Abort message to the coordinator) or w or q

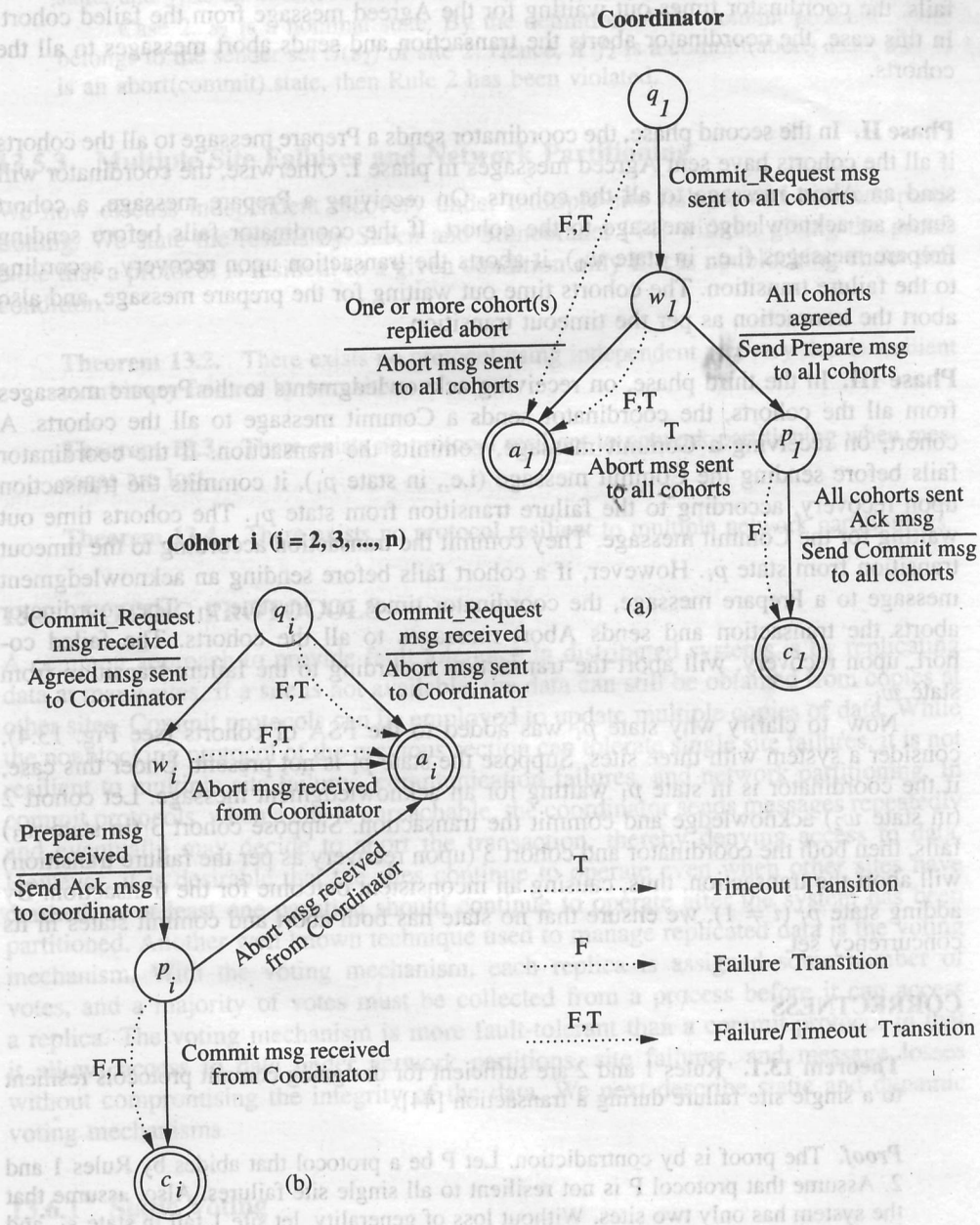

FIGURE 13.4
Finite state automata illustrating timeout and failure transitions (adapted from Skeen [44]).

depending on whether it has received the Commit_Request message or not. If a cohort fails, the coordinator times out waiting for the Agreed message from the failed cohort. In this case, the coordinator aborts the transaction and sends abort messages to all the cohorts.

Phase II. In the second phase, the coordinator sends a Prepare message to all the cohorts if all the cohorts have sent Agreed messages in phase I. Otherwise, the coordinator will send an Abort message to all the cohorts. On receiving a Prepare message, a cohort sends an acknowledge message to the cohort. If the coordinator fails before sending Prepare messages (i.e., in state w_1), it aborts the transaction upon recovery, according to the failure transition. The cohorts time out waiting for the prepare message, and also abort the transaction as per the timeout transition.

Phase III. In the third phase, on receiving acknowledgments to the Prepare messages from all the cohorts, the coordinator sends a Commit message to all the cohorts. A cohort, on receiving a Commit message, commits the transaction. If the coordinator fails before sending the Commit message (i.e., in state p_1), it commits the transaction upon recovery, according to the failure transition from state p_1. The cohorts time out waiting for the Commit message. They commit the transaction according to the timeout transition from state p_i. However, if a cohort fails before sending an acknowledgment message to a Prepare message, the coordinator times out in state p_1. The coordinator aborts the transaction and sends Abort messages to all the cohorts. The failed cohort, upon recovery, will abort the transaction according to the failure transition from state w_i.

Now, to clarify why state p_i was added to the FSA of cohorts (see Fig. 13.4), consider a system with three sites. Suppose the state p_i is not present. Under this case, if the coordinator is in state p_1 waiting for an acknowledgment message. Let cohort 2 (in state w_2) acknowledge and commit the transaction. Suppose cohort 3 (in state w_3) fails, then both the coordinator and cohort 3 (upon recovery as per the failure transition) will abort the transaction, thus, causing an inconsistent outcome for the transaction. By adding state p_i ($i \neq 1$), we ensure that no state has both abort and commit states in its concurrency set.

CORRECTNESS

Theorem 13.1. Rules 1 and 2 are sufficient for designing commit protocols resilient to a single site failure during a transaction [44].

Proof. The proof is by contradiction. Let P be a protocol that abides by Rules 1 and 2. Assume that protocol P is not resilient to all single site failures. Also, assume that the system has only two sites. Without loss of generality, let site 1 fail in state s_1, and let site 2 be in state s_2 when site 1 fails. Let site 1 make a failure transition to state f_1, and let site 2 make a timeout transition to state f_2. Suppose that the global state of the system, wherein site 1 is in state f_1 and site 2 is in state f_2, is inconsistent. Depending on whether s_2 is a final state (abort/commit) or a nonfinal state (all states other than abort and commit), we have the following two cases:

Case 1. s_2 is a final state. This implies that $f_2 \in C(s_1)$. If f_2 is a commit(abort) state, and f_1 is an abort(commit) state, then Rule 1 has been violated.

Case 2. s_2 is a nonfinal state. By the definition of the commit protocol, site 1 belongs to the sender set $S(s_2)$ of site 2. Hence, if f_2 is a commit(abort) state, and f_1 is an abort(commit) state, then Rule 2 has been violated.

13.5.3 Multiple Site Failures and Network Partitioning

We now discuss independent recovery under multiple site failures and network partitioning. We state the results by Skeen and Stonebraker [44] without giving the proof. Note that a protocol is resilient to a given condition only if it is nonblocking under that condition.

Theorem 13.2. There exists no protocol using independent recovery that is resilient to arbitrary failures by two sites.

Theorem 13.3. There exists no protocol resilient to network partitioning when messages are lost.

Theorem 13.4. There exists no protocol resilient to multiple network partitionings.

13.6 VOTING PROTOCOLS

A common approach to provide fault tolerance in distributed systems is by replicating data at many sites. If a site is not available, the data can still be obtained from copies at other sites. Commit protocols can be employed to update multiple copies of data. While the nonblocking protocol of the previous section can tolerate single site failures, it is not resilient to multiple site failures, communication failures, and network partitioning. In commit protocols, when a site is unreachable, the coordinator sends messages repeatedly and eventually may decide to abort the transaction, thereby denying access to data. However, it is desirable that the sites continue to operate even when other sites have crashed, or at least one partition should continue to operate after the system has been partitioned. Another well known technique used to manage replicated data is the voting mechanism. With the voting mechanism, each replica is assigned some number of votes, and a majority of votes must be collected from a process before it can access a replica. The voting mechanism is more fault-tolerant than a commit protocol in that it allows access to data under network partitions, site failures, and message losses without compromising the integrity of the data. We next describe static and dynamic voting mechanisms.

13.6.1 Static Voting

The static voting scheme is proposed by Gifford [19].

System model. The replicas of files are stored at different sites. Every file access operation requires that an appropriate lock is obtained. The lock granting rules allow