should update all replicas on a write request. We can ensure that it does so by maintaining a catalog table, which the system uses to determine all replicas for the data item.

19.3 Distributed Transactions

Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties (Section 14.1). There are two types of transaction that we need to consider. The **local transactions** are those that access and update data in only one local database; the **global transactions** are those that access and update data in several local databases. Ensuring the ACID properties of the local transactions can be done as described in Chapters 14, 15, and 16. However, for global transactions, this task is much more complicated, since several sites may be participating in execution. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

In this section, we study the system structure of a distributed database and its possible failure modes. In Section 19.4, we study protocols for ensuring atomic commit of global transactions, and in Section 19.5 we study protocols for concurrency control in distributed databases. In Section 19.6, we study how a distributed database can continue functioning even in the presence of various types of failure.

19.3.1 System Structure

Each site has its own *local* transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions. To understand how such a manager can be implemented, consider an abstract model of a transaction system, in which each site contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).
- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.

The overall system architecture appears in Figure 19.2.

The structure of a transaction manager is similar in many respects to the structure of a centralized system. Each transaction manager is responsible for:

- Maintaining a log for recovery purposes.

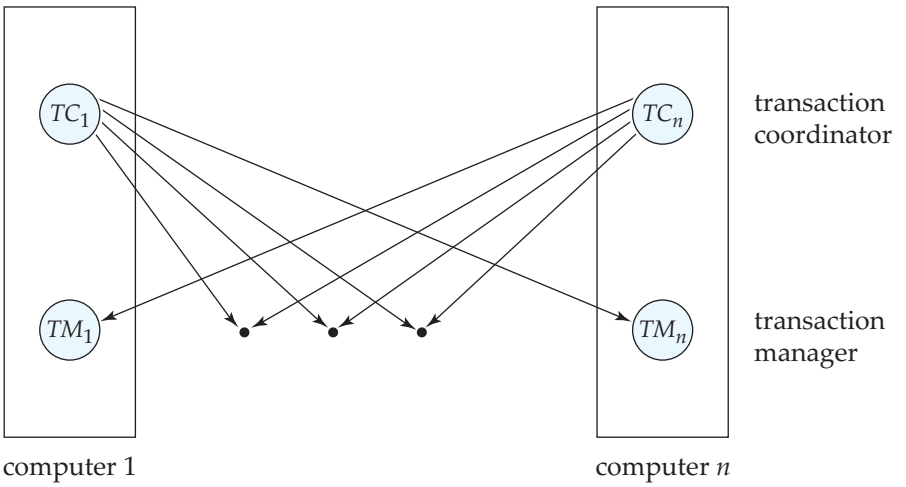

Figure 19.2 System architecture.

- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

As we shall see, we need to modify both the recovery and concurrency schemes to accommodate the distribution of transactions.

The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single site. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for:

- Starting the execution of the transaction.
- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

19.3.2 System Failure Modes

A distributed system may suffer from the same types of failure that a centralized system does (for example, software errors, hardware errors, or disk crashes). There are, however, additional types of failure with which we need to deal in a distributed environment. The basic failure types are:

- Failure of a site.
- Loss of messages.

- Failure of a communication link.
- Network partition.

The loss or corruption of messages is always a possibility in a distributed system. The system uses transmission-control protocols, such as TCP/IP, to handle such errors. Information about such protocols may be found in standard textbooks on networking (see the bibliographical notes).

However, if two sites A and B are not directly connected, messages from one to the other must be *routed* through a sequence of communication links. If a communication link fails, messages that would have been transmitted across the link must be rerouted. In some cases, it is possible to find another route through the network, so that the messages are able to reach their destination. In other cases, a failure may result in there being no connection between some pairs of sites. A system is **partitioned** if it has been split into two (or more) subsystems, called **partitions**, that lack any connection between them. Note that, under this definition, a partition may consist of a single node.

19.4 Commit Protocols

If we are to ensure atomicity, all the sites in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all sites, or it must abort at all sites. To ensure this property, the transaction coordinator of T must execute a *commit protocol*.

Among the simplest and most widely used commit protocols is the **two-phase commit protocol** (**2PC**), which is described in Section 19.4.1. An alternative is the **three-phase commit protocol** (**3PC**), which avoids certain disadvantages of the 2PC protocol but adds to complexity and overhead. Section 19.4.2 briefly outlines the 3PC protocol.

19.4.1 Two-Phase Commit

We first describe how the two-phase commit protocol (2PC) operates during normal operation, then describe how it handles failures and finally how it carries out recovery and concurrency control.

Consider a transaction T initiated at site S_i, where the transaction coordinator is C_i.

19.4.1.1 The Commit Protocol

When T completes its execution—that is, when all the sites at which T has executed inform C_i that T has completed—C_i starts the 2PC protocol.

- **Phase 1**. C_i adds the record <prepare T> to the log, and forces the log onto stable storage. It then sends a prepare T message to all sites at which T executed. On receiving such a message, the transaction manager at that site

determines whether it is willing to commit its portion of T. If the answer is no, it adds a record <no T> to the log, and then responds by sending an abort T message to C_i. If the answer is yes, it adds a record <ready T> to the log, and forces the log (with all the log records corresponding to T) onto stable storage. The transaction manager then replies with a ready T message to C_i.

- **Phase 2**. When C_i receives responses to the prepare T message from all the sites, or when a prespecified interval of time has elapsed since the prepare T message was sent out, C_i can determine whether the transaction T can be committed or aborted. Transaction T can be committed if C_i received a ready T message from all the participating sites. Otherwise, transaction T must be aborted. Depending on the verdict, either a record <commit T> or a record <abort T> is added to the log and the log is forced onto stable storage. At this point, the fate of the transaction has been sealed. Following this point, the coordinator sends either a commit T or an abort T message to all participating sites. When a site receives that message, it records the message in the log.

A site at which T executed can unconditionally abort T at any time before it sends the message ready T to the coordinator. Once the message is sent, the transaction is said to be in the **ready state** at the site. The ready T message is, in effect, a promise by a site to follow the coordinator's order to commit T or to abort T. To make such a promise, the needed information must first be stored in stable storage. Otherwise, if the site crashes after sending ready T, it may be unable to make good on its promise. Further, locks acquired by the transaction must continue to be held until the transaction completes.

Since unanimity is required to commit a transaction, the fate of T is sealed as soon as at least one site responds abort T. Since the coordinator site S_i is one of the sites at which T executed, the coordinator can decide unilaterally to abort T. The final verdict regarding T is determined at the time that the coordinator writes that verdict (commit or abort) to the log and forces that verdict to stable storage. In some implementations of the 2PC protocol, a site sends an acknowledge T message to the coordinator at the end of the second phase of the protocol. When the coordinator receives the acknowledge T message from all the sites, it adds the record <complete T> to the log.

19.4.1.2 Handling of Failures

The 2PC protocol responds in different ways to various types of failures:

- **Failure of a participating site**. If the coordinator C_i detects that a site has failed, it takes these actions: If the site fails before responding with a ready T message to C_i, the coordinator assumes that it responded with an abort T message. If the site fails after the coordinator has received the ready T message from the site, the coordinator executes the rest of the commit protocol in the normal fashion, ignoring the failure of the site.

  When a participating site S_k recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution

when the failure occurred. Let T be one such transaction. We consider each of the possible cases:

- The log contains a <commit T> record. In this case, the site executes redo(T).
- The log contains an <abort T> record. In this case, the site executes undo(T).
- The log contains a <ready T> record. In this case, the site must consult C_i to determine the fate of T. If C_i is up, it notifies S_k regarding whether T committed or aborted. In the former case, it executes redo(T); in the latter case, it executes undo(T). If C_i is down, S_k must try to find the fate of T from other sites. It does so by sending a querystatus T message to all the sites in the system. On receiving such a message, a site must consult its log to determine whether T has executed there, and if T has, whether T committed or aborted. It then notifies S_k about this outcome. If no site has the appropriate information (that is, whether T committed or aborted), then S_k can neither abort nor commit T. The decision concerning T is postponed until S_k can obtain the needed information. Thus, S_k must periodically resend the querystatus message to the other sites. It continues to do so until a site that contains the needed information recovers. Note that the site at which C_i resides always has the needed information.
- The log contains no control records (abort, commit, ready) concerning T. Thus, we know that S_k failed before responding to the prepare T message from C_i. Since the failure of S_k precludes the sending of such a response, by our algorithm C_i must abort T. Hence, S_k must execute undo(T).

- **Failure of the coordinator**. If the coordinator fails in the midst of the execution of the commit protocol for transaction T, then the participating sites must decide the fate of T. We shall see that, in certain cases, the participating sites cannot decide whether to commit or abort T, and therefore these sites must wait for the recovery of the failed coordinator.
  - If an active site contains a <commit T> record in its log, then T must be committed.
  - If an active site contains an <abort T> record in its log, then T must be aborted.
  - If some active site does *not* contain a <ready T> record in its log, then the failed coordinator C_i cannot have decided to commit T, because a site that does not have a <ready T> record in its log cannot have sent a ready T message to C_i. However, the coordinator may have decided to abort T, but not to commit T. Rather than wait for C_i to recover, it is preferable to abort T.
  - If none of the preceding cases holds, then all active sites must have a <ready T> record in their logs, but no additional control records (such

as <abort T> or <commit T>). Since the coordinator has failed, it is impossible to determine whether a decision has been made, and if one has, what that decision is, until the coordinator recovers. Thus, the active sites must wait for C_i to recover. Since the fate of T remains in doubt, T may continue to hold system resources. For example, if locking is used, T may hold locks on data at active sites. Such a situation is undesirable, because it may be hours or days before C_i is again active. During this time, other transactions may be forced to wait for T. As a result, data items may be unavailable not only on the failed site (C_i), but on active sites as well. This situation is called the **blocking** problem, because T is blocked pending the recovery of site C_i.

- **Network partition**. When a network partitions, two possibilities exist:
  1. The coordinator and all its participants remain in one partition. In this case, the failure has no effect on the commit protocol.
  2. The coordinator and its participants belong to several partitions. From the viewpoint of the sites in one of the partitions, it appears that the sites in other partitions have failed. Sites that are not in the partition containing the coordinator simply execute the protocol to deal with failure of the coordinator. The coordinator and the sites that are in the same partition as the coordinator follow the usual commit protocol, assuming that the sites in the other partitions have failed.

Thus, the major disadvantage of the 2PC protocol is that coordinator failure may result in blocking, where a decision either to commit or to abort T may have to be postponed until C_i recovers.

19.4.1.3 Recovery and Concurrency Control

When a failed site restarts, we can perform recovery by using, for example, the recovery algorithm described in Section 16.4. To deal with distributed commit protocols, the recovery procedure must treat **in-doubt transactions** specially; in-doubt transactions are transactions for which a <ready T> log record is found, but neither a <commit T> log record nor an <abort T> log record is found. The recovering site must determine the commit–abort status of such transactions by contacting other sites, as described in Section 19.4.1.2.

If recovery is done as just described, however, normal transaction processing at the site cannot begin until all in-doubt transactions have been committed or rolled back. Finding the status of in-doubt transactions can be slow, since multiple sites may have to be contacted. Further, if the coordinator has failed, and no other site has information about the commit–abort status of an incomplete transaction, recovery potentially could become blocked if 2PC is used. As a result, the site performing restart recovery may remain unusable for a long period.

To circumvent this problem, recovery algorithms typically provide support for noting lock information in the log. (We are assuming here that locking is used for concurrency control.) Instead of writing a <ready T> log record, the algorithm

writes a <ready T, L> log record, where L is a list of all write locks held by the transaction T when the log record is written. At recovery time, after performing local recovery actions, for every in-doubt transaction T, all the write locks noted in the <ready T, L> log record (read from the log) are reacquired.

After lock reacquisition is complete for all in-doubt transactions, transaction processing can start at the site, even before the commit–abort status of the in-doubt transactions is determined. The commit or rollback of in-doubt transactions proceeds concurrently with the execution of new transactions. Thus, site recovery is faster, and never gets blocked. Note that new transactions that have a lock conflict with any write locks held by in-doubt transactions will be unable to make progress until the conflicting in-doubt transactions have been committed or rolled back.

19.4.2 Three-Phase Commit

The three-phase commit (3PC) protocol is an extension of the two-phase commit protocol that avoids the blocking problem under certain assumptions. In particular, it is assumed that no network partition occurs, and not more than k sites fail, where k is some predetermined number. Under these assumptions, the protocol avoids blocking by introducing an extra third phase where multiple sites are involved in the decision to commit. Instead of directly noting the commit decision in its persistent storage, the coordinator first ensures that at least k other sites know that it intended to commit the transaction. If the coordinator fails, the remaining sites first select a new coordinator. This new coordinator checks the status of the protocol from the remaining sites; if the coordinator had decided to commit, at least one of the other k sites that it informed will be up and will ensure that the commit decision is respected. The new coordinator restarts the third phase of the protocol if some site knew that the old coordinator intended to commit the transaction. Otherwise the new coordinator aborts the transaction.

While the 3PC protocol has the desirable property of not blocking unless k sites fail, it has the drawback that a partitioning of the network may appear to be the same as more than k sites failing, which would lead to blocking. The protocol also has to be implemented carefully to ensure that network partitioning (or more than k sites failing) does not result in inconsistencies, where a transaction is committed in one partition and aborted in another. Because of its overhead, the 3PC protocol is not widely used. See the bibliographical notes for references giving more details of the 3PC protocol.

19.4.3 Alternative Models of Transaction Processing

For many applications, the blocking problem of two-phase commit is not acceptable. The problem here is the notion of a single transaction that works across multiple sites. In this section, we describe how to use *persistent messaging* to avoid the problem of distributed commit, and then briefly outline the larger issue of *workflows*; workflows are considered in more detail in Section 26.2.

To understand persistent messaging, consider how one might transfer funds between two different banks, each with its own computer. One approach is to have