action between client and server processes in a database system. A database transaction is represented by a sequence of synchronous request/reply operations that satisfy the atomicity, consistency, isolation, and durability (ACID) properties (to be described later). Transactions in communication are similar to transactions in databases except they are defined as a set of asynchronous request/reply communications that also have the ACID properties but are without the sequential constraint of the operations in a database transaction. A communication transaction may involve the multicast of the same message to replicated servers and different requests to partitioned servers. Transaction services and concurrency control for database transactions are discussed in Chapters 6 and 12. Here we only address the communication aspect of transactions.

4.3.1 The ACID Properties

The ACID properties are primarily concerned with achieving the concurrency transparency goal of a distributed system. In Section 2.2 we described concurrency transparency as a property that allows sharing of objects without interference. In a sense, the execution of a transaction appears to take place in a critical section. However, operations from different transactions are interleaved (in some “safe” way) to gain more concurrency. In addition, transactions have additional properties:

- **Atomicity**: Either all of the operations in a transaction are performed or none of them are, in spite of failures.
- **Consistency**: The execution of interleaved transactions is equivalent to a serial execution of the transactions in some order.
- **Isolation**: Partial results of an incomplete transaction are not visible to others before the transaction is successfully committed.
- **Durability**: The system guarantees that the results of a committed transaction will be made permanent even if a failure occurs after the commitment.

Since all four properties are related to consistency, sometimes it is preferable to call the second property serializability to differentiate it from the others. Atomicity refers to the consistency of replicated or partitioned objects. Violation of isolation is seeing something that has never occurred, and violation of durability is not seeing something that has actually occurred. Both are inconsistent perceptions of the system state.

Ensuring the ACID properties requires that the participating processors coordinate their execution of a transaction. We name the processor that initiates the transaction the *coordinator*, and we name the remaining processors the *participants*. A transaction begins with a multicast of requests from the coordinator to all the participants. The transaction ends with a final decision to *commit* or to *abort* the transaction, depending on whether or not the ACID properties can be satisfied. All participants must agree on the final decision. One solution to the atomicity requirement of a transaction is for each participant to defer making its operation permanent until being assured or informed that all other participants are also ready to do so. This technique is similar to the two stages, message receipt and delivery, that we saw in the two-phase protocol for total-order multicast in
Section 4.1.5. The following section describes how atomicity, isolation, and durability properties are achieved by the two-phase commit protocol for atomic transactions.

4.3.2 The Two-phase Commit Protocol

The two-phase commit (2PC) protocol is analogous to a real-life unanimous voting scheme. Voting is initiated by the coordinator of a transaction. All participants in the distributed transaction must come to an agreement about whether to commit or abort the transaction and must wait for the announcement of the decision. Before a participant can vote to commit a transaction, it must be prepared to perform the commit. A transaction is committed only if all participants agree and are ready to commit.

Each participant (including the coordinator) maintains a private workspace for keeping track of updated data objects. Each update contains the old and new value of a data object. Updates will not be made permanent until the transaction is finally committed, to ensure the isolation semantics of the transactions. To cope with failures, it is necessary to flush the updates to a stable storage. The updates recorded in the stable storage are an activity log of a transaction. Each participating site has an activity log. The activity log can be replayed upon the recovery of a failure to facilitate either the redo of committed transactions or the undo of uncommitted transactions. A stable activity log is necessary for the durability or permanence of a committed transaction.

Figure 4.15 illustrates the execution flow of a two-phase commit atomic transaction. There are two synchronization points, precommit and commit, for each participating site. The coordinator begins a transaction by writing a precommit record into its activity log. The coordinator must be prepared to commit the transaction before writing the precommit record (i.e., updates have been flushed to stable log, resources are available to perform the commit, etc.). Writing the pre-commit record into the activity log tells the coordinator the status of the transaction if a failure occurs; the transaction has finished execution but is not yet committed. A vote request is then multicast to all participants. When a participant receives the vote request, it tests whether the transaction can be committed (the updates have been flushed to the activity log, serializability is ensured, resources are available, etc.). If the test is positive, the participant writes a precommit into the log and sends a YES reply to the coordinator. Otherwise, the participant aborts the transaction and sends a NO reply to the coordinator.

If the coordinator is able to collect all YES replies within a time-out interval, it commits the transaction by writing a commit record in the log and multicasting a commit message to all participants. Otherwise the coordinator aborts the transaction and multicasts an abort message. On receiving the commit message, each participant commits the transaction by writing a commit record into the activity log and releasing the transaction's resources. Finally, any response is returned to the coordinator. If the received message was an abort, the participant writes an abort record into the log, aborts the transaction, and releases the transaction's resources.

When used with an activity log, the two-phase commit protocol is highly resilient to processor failures. Figure 4.16 shows the time-lines of the protocol for the coordinator and a participant. Since writing precommit and commit to the activity log flushes all updates before these synchronization points, proper actions upon recovery from failures
can rely on the replay of the log at least up to the synchronization points. Thus recovery actions can be categorized into three types: failures before a precommit, failures after a precommit but before a commit, and failures after a commit. A processor (either the coordinator or a participant) can simply abort the transaction if it discovers from the log that the failure occurred before a precommit. This is equivalent to voting NO for the transaction. The coordinator can also abort the transaction if it crashes between precommit and commit, but it is more efficient to attempt to commit the transaction by remulticasting the request messages (if duplicates can be detected by the participants).

**Figure 4.16** Failures and recovery actions for the 2PC protocol.

Coordinator failure recovery actions

<table>
<thead>
<tr>
<th>begin</th>
<th>pre-commit request</th>
<th>send</th>
<th>collect</th>
<th>commit</th>
<th>send</th>
<th>receive</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>abort or continue</td>
<td>ressend</td>
<td>commit message</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>begin</th>
<th>receive pre-commit request</th>
<th>send</th>
<th>reply</th>
<th>receive commit</th>
<th>send</th>
<th>resp. update</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>find out commit or abort</td>
<td>continue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Similarly, it is safe to resend the commit messages if the coordinator crashes after writing the commit record to its log. The recovery action is slightly more complicated if a participant crashes between precommit and commit. The recovering participant must determine whether the transaction was committed or aborted by contacting the coordinator or the other participants. Finally if a participant recovers from a failure after a commit record has been written to the log, the participant simply makes the transaction’s updates permanent. With stable storage, the recovery mechanisms ensure the durability of the commitments.

Section 12.1.1 contains a more formal description of the two-phase commit protocol and its enhancement: the three-phase commit protocol. Chapter 13 is devoted to failure and recovery handling.

4.4 NAME AND DIRECTORY SERVICES

A discussion of communication is not complete without considering naming and addressing issues. Making a request to a service or accessing an object by means of interprocess communication requires that one first locate the service or object. Services are abstractions of objects. They are usually represented by processes with a service access point. Objects may be users, computers, communication links, or other resources such as files. Services and objects are normally identified by textual names. Alternatively, if names are unknown, service or object entities can be described by using attributes associated with them. Although services and objects have distinct meanings, their naming issues are similar. For clarity, we will use the term object entity to represent both services and objects in the discussion of name and directory services.

4.4.1 Name and Address Resolution

Name and directory services, in a narrow sense, are look-up operations. Given the name or some attributes of an object entity, more attribute information is obtained. The terms name service and directory service are often used interchangeably. A name service is a generic way of describing how a named object can be addressed and subsequently located by using its address. The term directory service is sometimes used to describe a special name service, such as the directory service of a file system. Other times, it is used to represent a very general name service for all kinds of attribute look-ups on different object types, not just limited to address information. X.500, defined by CCITT, is an example of such a directory service. Higher-level name services can be built on top of a standard directory service.

Any object entity in a system must be named (or be identifiable) and located before it can be used. The operation of locating an object, called a resolution process, may require a number of look-up (or mapping) operations. Each object entity has a logical address in the operating system and a physical location in the network. The resolution process involves two stages: a name resolution that maps names to logical addresses and an address resolution that maps logical addresses to network routes.
a multicast, which is logically a single operation. Messages in the two-phase multicast protocol are ordered by multicast completion time. If we were to use the same strategy for transactions, updates would have to be ordered by transaction completion time. This means that transactions are processed in batches. The delay of one transaction will tie up the others, making the protocol impractical for transactions.

We have illustrated the concept of serializability by using a simple two-transaction example. For systems with a larger number of interacting transactions, a more precise model for serializability is needed. Chapter 12 presents the serialization graph model to address general serialization problems.

6.3.3 Concurrency Control Protocols

There are three general approaches to the concurrency control problem, whereby inconsistency can be prevented or avoided or consistency can be validated. In this section, we discuss two-phase locking, timestamp ordering, and optimistic concurrency control, which are examples of these approaches.

Two-phase Locking

Using the locking approach, all shared objects in a well-formed transaction must be locked before they can be accessed and must be released before the end of the transaction. The two-phase locking (2PL) protocol adds an additional requirement that a new lock cannot be acquired after the first release of a lock. A transaction is divided into two phases: a growing phase of locking and a shrinking phase of releasing the objects. An extreme case of a 2PL transaction is one that locks all objects in the database at the beginning of the transaction and releases all of them at the same time at the end of the transaction. Serialization is trivial because the only possible schedules are serial schedules. This method is used by some simple database applications but is not generally acceptable since it completely ignores sharing and concurrency. One can increase concurrency by releasing locks as soon as possible. However, typical practice is to release locks at the commit point, discussed shortly.

Two-phase locking produces serializable transactions, as can be seen from the above example. If transaction $t_1$ has acquired object C in operation 1, it will not release the object until after operation 2 has been completed. This will make it impossible for operation 3 in transaction $t_2$ to occur between operations 1 and 2. Similarly, if transaction $t_2$ obtains the lock on C first, operation 1 in transaction $t_1$ cannot be scheduled between operations 3 and 4. Updates to D are forced to have the same order as updates to C. This means that schedules 3, 4, 5, and 6 in Table 6.1 are not feasible if $t_1$ and $t_2$ are 2PL transactions. The resulting schedules are limited to schedules 1 and 2. Thus, two-phase locking sacrifices concurrency for serializability.

Locking a resource and waiting for another is a source of potential deadlocks in any system. If operations 3 and 4 are reversed in transaction $t_2$, it is possible that transaction $t_1$ may be holding C and waiting for D while transaction $t_2$ is holding D and waiting for C. The result is a circular hold-and-wait deadlock.

It would seem that the TM should attempt to release locks as soon as possible; as soon as it knows that the last lock has been obtained and a data item will not be accessed by
the transaction again. However, there are several difficult problems with this approach. First, we may encounter rolling aborts. Suppose that transaction $t_1$ updates data item C, and then releases the lock on C. Suppose next that transaction $t_2$ reads the value in C. If transaction $t_1$ aborts, $t_2$ must abort also, because $t_2$ has read dirty data. The abort of $t_2$ might cause the abort of another transaction $t_3$, and so on. Since $t_2$ can commit only if $t_1$ can commit, $t_2$ has a commit dependence on $t_1$. The commit dependencies must be tracked, and the commit of $t_2$ must be delayed until after the commit of $t_1$.

The problems of rolling aborts and commit dependence tracking can be solved by using strict two-phase locking. A transaction only releases its locks at its commit or abort point. It is difficult to implement two-phase locking systems that are not strict, since the TM does not know when the last lock has been requested. Strict two-phase locking sacrifices some concurrency but has the benefit of greatly simplifying the implementation.

**Timestamp Ordering**

Two-phase locking orders conflicting operations by the time a shared object is first locked. Previously we have seen many applications of logical timestamps for event ordering. Conflicting operations in interleaved transactions can be ordered by timestamps as well. If every object manager follows the transaction timestamp order to perform its invoked operations, the execution of transactions satisfies the serializability conditions.

Let us assume that $t_0$, $t_1$, and $t_2$ in the previous example are unique logical timestamps representing the three transactions, such that $t_0 < t_1 < t_2$. (We are using $t_1$ to represent both the transaction and its timestamp.) When an operation on a shared object is invoked, the object records the timestamp of the invoking transaction. Suppose that later, a different transaction invokes a conflicting operation on the object. If the transaction has a larger timestamp than the one recorded by the object, we let the operation proceed (and record the new timestamp). If the transaction has a smaller timestamp, we must abort it; otherwise the transaction will execute out of order. The aborted transaction restarts and samples a larger timestamp.

Allowing a conflicting operation to proceed means that the operation is permitted to wait for its turn at the object site (i.e., an implicit lock). Younger (larger timestamp) transactions wait for older (smaller timestamp) transactions and older transactions die and restart when they confront with a younger transaction. Deadlocks are not possible because the execution of transactions based on the increasing order of transaction timestamps excludes the possibility of a circular hold-and-wait condition. Using this timestamp ordering approach, schedules 1 and 3 in Table 6.1 are feasible schedules and satisfy the consistency constraint. For schedules 2, 4, and 6, transaction $t_1$ arrives too late at object C. Transaction $t_1$ is aborted before it accesses object D. It will return with a larger timestamp, and will be executed in a schedule similar to that of 1 and 3.

The situation for schedule 5 is slightly more complicated. Operations 1, 3, and 4 execute without a problem, but operation 2 on object D in transaction $t_1$ sees a larger timestamp written by transaction $t_2$ in object D. To abort transaction $t_1$, we must also nullify its operation on C. In this particular example, object C, written by $t_1$, is simply rewritten by $t_2$'s operation 3, causing no effect on $t_2$. However, if operation 3 were a read operation, $t_2$ would have read some data that is not supposed to be visible to it.