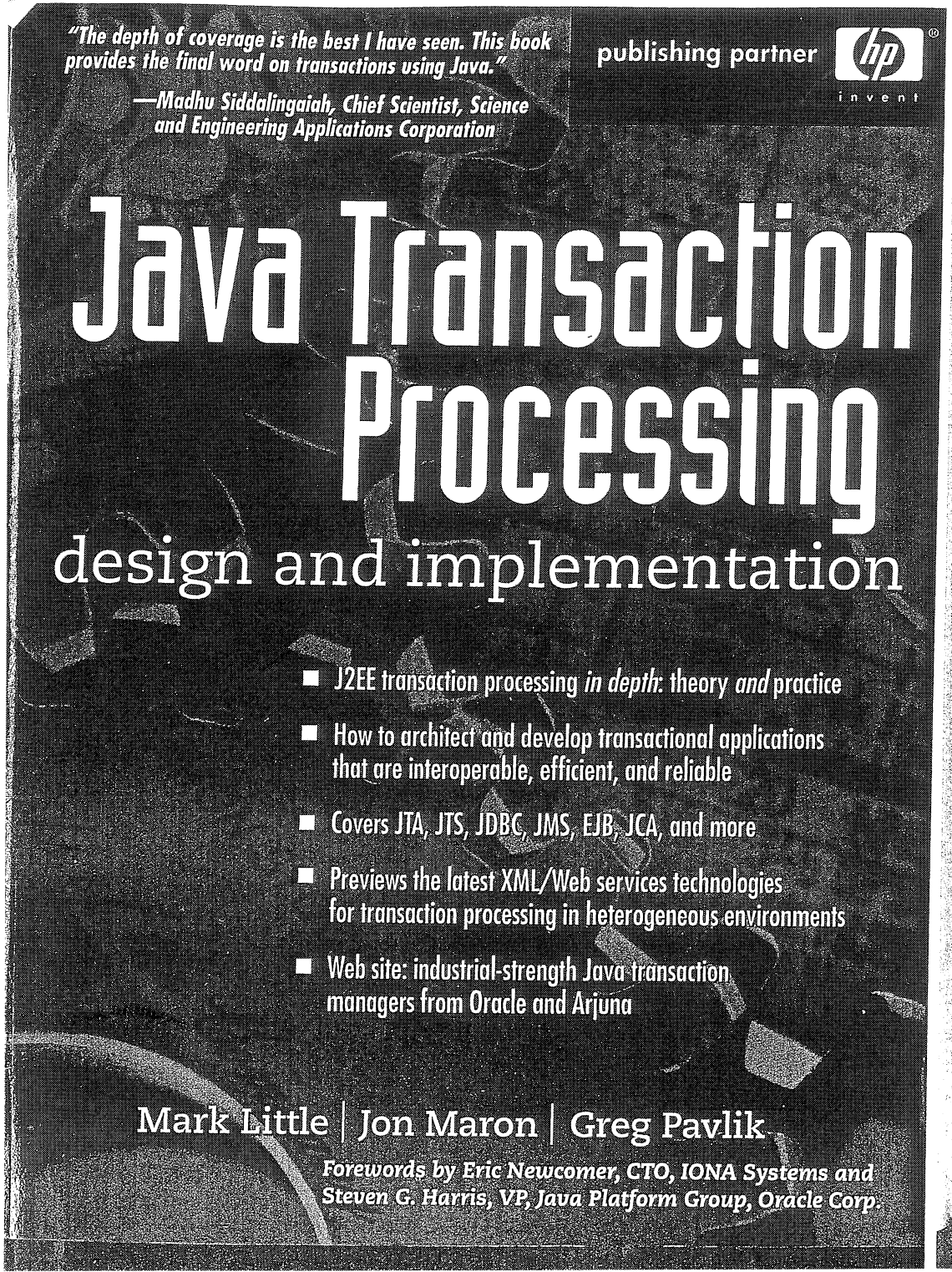

CHAPTER 1

Transactions Fundamentals

In order to fully understand the issues involved in designing transaction systems and using transactions in applications that will be covered in subsequent chapters, it's necessary to have some understanding of the fundamental principles behind transactions. For example, to understand the effect that distributed transactions have on your EJB application it's obviously necessary to know what a "distributed transaction" is and how it is likely to be implemented.

Transaction processing systems and the theoretical and implementation principles behind them have been around in one form or another for nearly five decades. Describing all of these various principles in detail would require a book in its own right and unfortunately we don't have the space. Fortunately we don't need to cover all of these principles in great detail. However, we attempt to provide enough information that you won't have to go elsewhere for answers to questions this book might raise.

> What this chapter does is lay the groundwork for all of the other chapters, providing readers with little or no transaction knowledge with enough information to place the rest of the book in context. If you are an experienced transaction user then you may be able to skim through this chapter, although you may still find some interesting new facts. For a more detailed treatment of many of these topics we would recommend other texts such as *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter and *Principles of Transaction Processing* by Philip Bernstein and Eric Newcomer.

What Are Transactions and Why Do I Need Them?

Imagine that you are implementing a centralized (single machine) bank application, where the information about each bank account is stored in a persistent database. Let's also assume that each account is managed separately within the database. Now, your users won't be very happy if their money gets lost because of failures. For example, your bank has to allow the transfer of money from one account to another, but obviously a failure after the money has left the original account and before it has been placed into the new account could mean that the money is lost in the system! What you would like is an all-or-nothing operation: the money either moves between the two accounts or it stays in the first account. This would be referred to as an *atomic operation*: such an operation is indivisible, i.e., it is not possible to decompose it into smaller operations.

It's obviously difficult enough to consider how to provide this functionality in a centralized system, so you can imagine the additional problems that are presented when we add multiple machines or components to the application. Distributed systems pose reliability problems not frequently encountered in more traditional centralized systems. A distributed system consisting of a number of computers connected by some communication network is subject to independent failure modes of its components, such as nodes, links and operating systems. What this means is that the lack of any centralized control can mean that part of the system can fail (e.g., a machine may crash or a network connection cease to work) while other parts are still functioning, thus leading to the possibility of abnormal behavior of application programs in execution. This explains why one definition of a distributed system is "any system where the failure of a machine you have never heard of prevents you from doing work."

If you had to consider all possible failure scenarios when designing and building a distributed application, it is unlikely that you'd ever finish. Imagine all of the "what if?" conditions that might arise; for example, "what if the client crashes in the middle of making a request to a server?" or "what if the server crashes while updating the back-end database?" and "if the system recovers, how do I restart the application and ensure those invocations in progress when the failure occurred continue or are undone?" Asking all programmers to have to consider such scenarios whenever they build applications is impractical and that's why there has been a lot of work over the years involved in building tools to assist in the construction of fault-tolerant distributed systems.

One of those tools is the *atomic transaction*. As we see in the following sections, an atomic transaction (often abbreviated to *transaction*) provides an "all-or-nothing" property to work that is conducted within its scope, also ensuring that shared resources are protected from multiple users. Importantly, application programmers typically only have to start and end a transaction; all of the complex work necessary to provide the transaction's properties is hidden by the transaction system, leaving the programmer free to concentrate on the more functional aspects of the application at hand.

Let's take a look at just how a transaction system could help in a real-world application environment. Consider the case of a distributed system (shown in Figure 1-1) where each

machine provides various services (e.g., data storage, printing, bank account) that can be invoked by an application program. It is natural for an application that uses a collection of these services to require that they behave consistently in the presence of failures. In the figure the bank offers two kinds of accounts to users: a CurrentAccount and a SavingsAccount. Both of these accounts are maintained on persistent (durable) storage, in this case represented by a database, with suitable front ends (represented as services in the figure). Finally there is a transaction manager service that will be used to manage any transactions that the bank may require in order to process the user's requests. We see later why the transaction manager requires a log, but in brief it's to ensure that transactions can tolerate failures.

Let's consider a simple example: imagine an online bank that allows customers to transfer funds between accounts (CurrentAccount and SavingsAccount), and let's further assume that Mr. Smith has both of these accounts whose balances are $500 and $1,400, respectively. As shown, the bank is responsible for communicating with the transaction service to start and end the transaction Mr. Smith will use to manage transferring his funds.

Mr. Smith wants to transfer $400 from the CurrentAccount to the SavingsAccount and because of bank processes, this will occur in the following steps:

1. Read the amount of money in the CurrentAccount to determine that there are enough funds to transfer.
2. Debit $400 from the CurrentAccount.
3. Credit $400 to the SavingsAccount.
4. The bank checks that the CurrentAccount is still in credit and if not, charges Mr. Smith for an overdraft. This last step is just in case there are concurrent accesses to the Smith account. As we see later, the use of atomic transactions can make this step redundant, but without them something along these lines will obviously be required.

The transfer process may be affected by failures of software or hardware that could affect the overall consistency of the system in a number of ways. For example, if a failure occurs between step 2 and 3 then it is entirely possible for the $400 to be removed from the CurrentAccount and to vanish into the ether rather than be credited to the SavingsAccount. This loss of data integrity would have dire consequences for the user and ultimately the bank.

Now let's assume that Mrs. Smith wants to withdraw $200 from the CurrentAccount to go shopping. The steps her withdrawal goes through are:

1. Check the CurrentAccount has sufficient funds.
2. Withdraw $200.
3. The bank checks that the CurrentAccount is still in credit and if not, charges Mr. Smith for an overdraft.

If Mrs. Smith withdraws at the same time her husband is transferring funds, then it is entirely possible for both operations to see sufficient funds in the CurrentAccount for their own

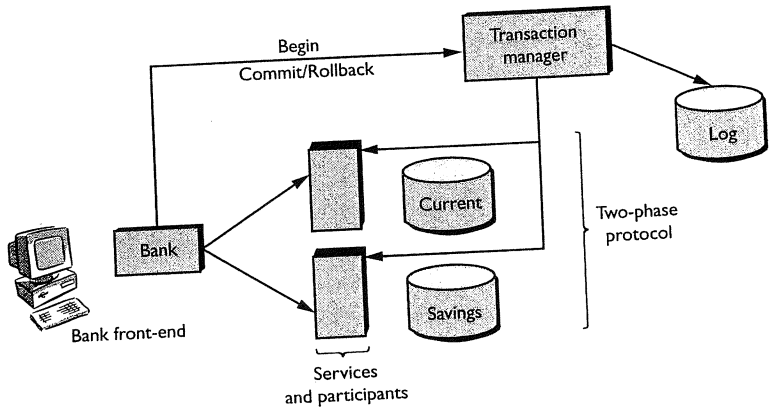

Figure 1-1 Bank account transfers.

requirements, when in fact there are insufficient funds for both. The result is that the CurrentAccount ends up $100 overdrawn and Mr. Smith gets an overdraft charge he didn't expect.

What Mr. Smith would like is that accesses to the account are handled in such a manner that consistency of both accounts is maintained despite failures or concurrent access. And this is in fact a more general statement of desirability for distributed applications that manipulate data or shared resources. Fortunately, atomic transactions (transactions) have the following properties that make them ideal for solving this kind of problem (these are often referred to as ACID properties):

- *Atomicity*: The transaction completes successfully (commits) or if it fails (aborts) *all* of its effects are undone (rolled back). In the bank example, Mr. Smith wants his money to move between the two accounts or for it to remain in the CurrentAccount; the atomic property would guarantee this.
- *Consistency*: Transactions produce consistent results and preserve application-specific invariants (preserves the internal consistency of the data it acts on). The consistency of Mr. Smith's account data is maintained.
- *Isolation*: Intermediate states produced while a transaction is executing are not visible to others. Furthermore, transactions appear to execute *serially*, even if they are actually executed concurrently. This property would ensure that Mr. Smith and Mrs. Smith's bank operations don't conflict.
- *Durability*: The effects of a committed transaction are never lost (except by a catastrophic failure). As we saw earlier, the bank account information is maintained on some external database and the transaction ensures that it is updated correctly.

A transaction can be terminated in two ways: *committed* or *aborted* (rolled back). When a transaction is committed, all changes made within it are made durable (forced onto stable storage, e.g., disk). When a transaction is aborted, all of the changes are undone. As we shall see later, atomic transactions can also be nested; the effects of a nested action are provisional upon the commit/abort of the outermost (*top-level*) atomic transaction.

Associated with every transaction is a *coordinator*, which is responsible for governing the outcome of the transaction. The coordinator may be implemented as a separate service or may be co-located with the user for improved performance. It communicates with enlisted participants to inform them of the desired termination requirements, i.e., whether they should accept (*commit*) or reject (*rollback*) the work done within the scope of the given transaction. For example, it communicates whether to purchase the (provisionally reserved) flight tickets for the user or to release them. A *transaction manager factory* is typically responsible for managing coordinators for many transactions. The initiator of the transaction (e.g., the client) communicates with a transaction manager and asks it to start a new transaction and associate a coordinator with the transaction.

In the following sections, we examine each of these properties in more detail.

Atomicity

In order to ensure that a transaction has an atomic outcome, a multi-phase commit protocol is required. The industry standard transaction protocols use *two-phase commit*. (Some transaction protocols support a three-phase termination protocol, which tries to reduce the chances of blocking due to the failure of the coordinator, but two-phase is the most popular protocol in practice.) This protocol is used to guarantee consensus between participating members of the transaction. Figure 1-2 illustrates the main aspects of this protocol: during phase 1, the transaction coordinator, C, attempts to communicate with all of the action participants, A and B, to determine whether they will commit or abort. An abort reply, or no reply, from any participant acts as a veto, causing the entire action to abort.

> **NOTE** The failure to deliver the initial prepare request may also cause the coordinator to roll back (possibly after retrying to deliver the message some number of times).

Based upon these (lack of) responses, the coordinator arrives at the decision of whether to commit or abort the transaction. If the transaction will commit, the coordinator records this decision on stable storage, and the protocol enters phase 2, where the coordinator forces the participants to carry out the decision. The coordinator also informs the participants if the transaction aborts.

> **NOTE** Failure to write the intention should also cause the transaction to roll back.

When each participant receives the coordinator's phase 1 message, they record sufficient information on stable storage to either commit or abort changes made during the action. After

returning the phase 1 response, each participant that returned a commit response *must* remain blocked until it has received the coordinator's phase 2 message. Until they receive this message, these resources are unavailable for use by other actions. If the coordinator fails before delivery of this message, these resources remain blocked. However, if crashed machines eventually recover, crash recovery mechanisms can be employed to unblock the protocol and terminate the transaction.

If the client decides to roll back the transaction then the coordinator won't run a two-phase protocol: the participants will simply be told to roll back.

> **NOTE** The two-phase commit protocol is not client-server based. It simply talks about a coordinator and participants, and makes no assumption about where they are located. Different implementations of the protocol may impose certain restrictions about locality, but these are purely implementation choices.

You might wonder what happens if during the second phase A or B decide that they cannot commit. In the strict two-phase protocol, this is not allowed: once participants have agreed to a decision during the first phase, this is considered a *guarantee* and they cannot deviate from this promise. However, in practice this doesn't always happen and the two-phase protocol on which most transaction systems are based allows participants to make different choices during phase 2 than were made during phase 1 *in exceptional circumstances* (e.g., a critical failure occurs which means that the participant simply cannot ever commit). As we see later, this can result in non-atomic behavior and gives what are commonly known as *heuristic outcomes*.

When running the first phase of the commit protocol, it's not strictly necessary for each participant to be sent the prepare message sequentially: the coordinator could invoke each participant in parallel to improve performance and collate all responses to then decide whether to commit or roll back the transaction. In fact, some commercial transaction systems support this option (e.g., the Arjuna Transaction Service).

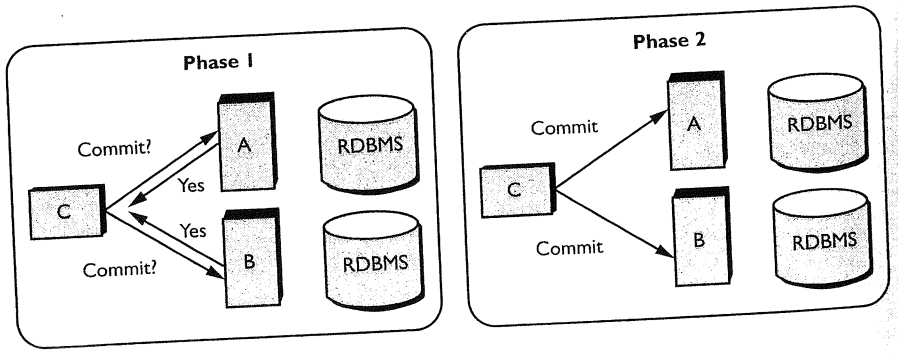

Figure 1-2 Two-phase commit protocol.

Consistency

A transactional application should maintain the consistency of the resources (e.g., databases, file-systems, and so on) that it uses. In essence, transactional applications should move from one consistent state to another. However, unlike the other transactional properties (A, I and D), this is something that the transaction system cannot achieve by itself since it does not possess any semantic information about the resources it manipulates: it would be impossible for a transaction processing system to assert that the resources are moving to (or from) consistent states. Take our previous example of Mr. Smith's bank accounts: how would a transaction system know that the accounts should never go into the red? All a transaction system can ensure is that any state changes that do occur are performed in a manner that is guaranteed despite failures. It is the application programmers' responsibility to ensure consistency (in whatever way makes sense for the application concerned).

Isolation (Serializability)

If we assume that objects and services can be shared between various programs, it is necessary to ensure that concurrent executions of programs are free from interference, i.e., concurrent executions should be equivalent to some serial order of execution. We have already seen a fairly trivial example of this through the online bank account, where Mr. and Mrs. Smith conflict over their bank accounts, but it is worth formalizing this requirement.

Let's try to illustrate what we mean by a serializable execution. Consider the following two programs (where *w, x, y* and *z* are distinct state variables) and P1 and P2 are programs that perform the specified operations on those variables:

$$P1 : z := 10; x := x+1; y := y+1$$

$$P2 : w := 7: x := x * 2; y := y * 2$$

If we assume that $x=y=2$ initially, then a serial execution of P1 followed by P2 (P1;P2) will produce the result z=10, w=7, x=y=6, and execution P2;P1 will produce the results z=10, w=7, x=y=5.

However, if you look at P1 and P2 you can see that *z* and *w* are variables that are not shared between the two programs. So it should be possible to execute some aspects of P1 and P2 concurrently. The partly concurrent execution order given below (indicated by the ||, which means that the operations on *z* and *w* execute in parallel) will be termed *interference free* or *serializable*, since it is equivalent to the serial order P1;P2, i.e., it produces the same results:

$$(z := 10 \,||\, w := 7); x := x+1; y := y+1; x := x * 2; y := y * 2$$

However, an execution order such as the one below is not free from interference since it cannot be shown to be equivalent to any serial order.

$$(z := 10 \parallel w := 7); x := x+1; x := x * 2; y := y * 2; y := y+1$$

Programs that possess the above-mentioned serializable property are said to be *atomic with respect to concurrency*. What this means is that the execution of a transaction is serializable from the perspective of an external user if it appears to that user as though that transaction runs by itself (exclusively). Even though multiple transactions may be executing concurrently, it is not possible for the external user to see this, if those transactions are serializable.

Combined with the consistency property we've already mentioned, any serial execution of these transactions will continue to preserve consistency of the data that is manipulated by the transactions. If we only had the consistency attribute, then executing transactions in parallel would not result in a globally consistent system.

Obviously this says nothing about durability (persistence) of these computations or the effect of failures on them, e.g., a failure may occur part way through a computation or in such a manner that all data is lost (remember the bank account?). However, the serializability property is extremely important, especially in an environment where multiple concurrent users may be attempting to use the same resources consistently. With this property, Mr. and Mrs. Smith can at least rest assured that neither will be able to push the account into the red (at least not unintentionally).

One obvious way of maintaining serializability would be to execute all transactions serially. However, that's obviously a very inefficient approach, especially if some operations within transactions can logically occur in parallel, as we saw above. So, to ensure serializability, transactional systems like relational databases typically make use of concurrency control mechanisms.

Two-Phase Concurrency Control

In this section we briefly discuss the topic of concurrency control for transactions. A very simple and widely used approach is to regard all operations on objects to be of type "read" or "write," which follow the synchronization rule permitting concurrent "reads" but exclusive "writes." This rule is imposed by requiring that any computation intending to perform an operation that is of type read (write) on an object, first acquire a "read lock" ("write lock") associated with that object. A read lock on an object can be held concurrently by many computations provided no computation is holding a write lock on that object. A write lock on an object, on the other hand, can only be held by a computation provided no other computation is holding a read or a write lock.

NOTE Although we talk in terms of locks, this should not be used to infer a specific implementation. Timestamp-based concurrency control could just as easily be used, for example.

In order to ensure the atomicity property, all computations must follow a "two–phase" locking policy, as illustrated in Figure 1-3. During the first phase, termed the growing phase, a computation can acquire locks, but not release them. The tail end of the computation constitutes the shrinking phase, during which time held locks can be released but no locks can be acquired. Now suppose that a computation in its shrinking phase is to be rolled back, and that some objects with write locks have already been released. If some of these objects have been locked by other computations, then abortion of the computation will require these computations to be aborted as well. To avoid this *cascade roll back* problem, it is necessary to make the shrinking phase "instantaneous," as shown by the dotted lines. In effect this means that all the held locks are released simultaneously.

Optimistic Versus Pessimistic Concurrency Control

Most transaction systems utilize what is commonly referred to as *pessimistic concurrency control* mechanisms: in essence, whenever a data structure or other transactional resource is accessed, a lock (for example) is obtained on it as described earlier. This lock will remain held on that resource for the duration of the transaction. The benefit of this is that other users will not be able to modify (and possibly not even observe) the resource until the holding transaction has

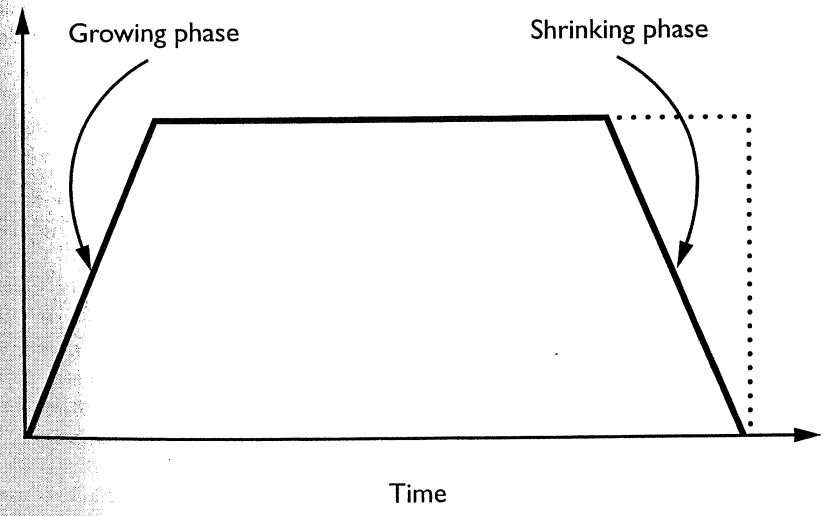

Figure 1-3 Two-phase locking.

terminated. There are a number of disadvantages of this style: (1) the overhead of acquiring and maintaining concurrency control information in an environment where conflict or data sharing is not high; (2) deadlocks may occur, where one user waits for another to release a lock not realizing that that user is waiting for the release of a lock held by the first (we talk about deadlocks in more detail later); (3) concurrency may be limited by the granularity of the locks; for example, page-level locking sometimes occurs in systems where multiple objects reside within the same page, and obtaining a lock on one object means locking them all, even if the objects are really independent.

Therefore, *optimistic concurrency control* assumes that conflicts are not high and tries to ensure locks are held only for brief periods of time: essentially locks are only acquired at the end of the transaction when it is about to terminate. This kind of concurrency control requires a means to detect if an update to a resource does conflict with any updates that may have occurred in the interim and how to recover from such conflicts. Typically, detection will happen using timestamps, whereby the system takes a snapshot of the timestamps associated with resources it is about to use or modify and compares them with the timestamps available when the transaction commits.

Resolution of conflicts is a different problem entirely, since in order to do so requires semantic information about the resources concerned (exactly what does it mean to find that the state of Mr. Smith's CurrentAccount has changed because Mrs. Smith got her transaction in first?). Therefore, most transaction systems that offer optimistic schemes will typically cause the detecting transaction to roll back and the application must retry, this time with a fresh copy of the data. Obviously this may result in a lot of work being lost, especially if the transaction that rolls back has been running for some time.

Assuming both optimistic and pessimistic concurrency control are available to you (and they may not be), then which one to use is up to you. A close examination of the environment in which the application and transactional resources reside is necessary to determine whether (a) shared access to resources occurs, and (b) the relative probability that sharing will cause a transaction to roll back. This might very well not be a black or white choice and may change over the lifetime of your objects or application. Certainly the use of different concurrency control schemes can be important when trying to improve the throughput of user requests and committed transactions, so it's well worth considering and understanding the issues involved. Optimistic and pessimistic locking strategies are often available in both database systems and middleware application servers, as we discuss in more detail later.

Type-Specific Concurrency Control

Another possible enhancement is to introduce *type-specific concurrency control*, which is a particularly attractive means of increasing the concurrency in a system by reducing the granularity of the locks used to achieve isolation. Concurrent read/write or write/write operations are permitted on an object from different transactions provided these operations can be shown to be non-interfering (for example, for a directory object, reading and deleting different entries can be permitted to take place simultaneously). Object-oriented systems are well suited to this approach, since semantic knowledge about the operations of objects can be exploited to control permissible concurrency within objects. Additional work may be needed when working with procedural systems.

Deadlock Detection and Prevention

When multiple transactions compete for the same resources in conflicting modes (locks), it is likely that some of them will fail to acquire those resources. If a transaction that cannot acquire a lock on a resource waits for it to be released, then that transaction is *blocked*—no forward progress can be made until the lock has been acquired. In some environments, it is possible for some transactions to be waiting for each other, where each of them is blocked and is also blocking another transaction. In this situation, none of the transactions can proceed and the system is *deadlocked*.

For example, let's consider two transactions T_1 and T_2 that operate on two resources X and Y. Let's assume that the execution of the operations involved in these transactions is:

- T_1: read(X); write(Y)
- T_2: read(Y); write(X)

If the serial execution of these transactions were to result in:

- readT_1(X); readT_2(Y); writeT_2(X); readT_1(Y)

NOTE readT_1 means the read operation performed by T_1, etc.

Assume that T_1 obtained a read lock on X and then T_2 gets a read lock on Y—possible because these operations aren't conflicting and can thus occur in parallel. However, when T_2 comes to write to X its attempt to get a write lock on X will be blocked because T_1 still holds its read lock. Likewise, T_1's attempt to get a write lock on Y will block because of the read lock that T_2 holds. Each transaction is blocked waiting for the release of the other's read lock before they can progress: they are *deadlocked*.

The only way for the deadlock to be resolved is for at least one of the transactions to release its locks that are blocking another transaction. Obviously such a transaction cannot commit (it has not been able to perform all of its work since it was blocked); therefore, it must roll back.

Deadlock detection and prevention is complicated enough in a non-distributed environment without then including the extra complexity of distribution. In general, most transaction systems allow deadlocks to occur, simply because to do otherwise can be too restrictive for applications. There are several techniques for deadlock detection, but the two most popular are:

- *Timeout-based*: If a transaction has been waiting longer than a specified period of time, the transaction system will automatically roll back the transaction on the

assumption it is deadlocked. The main advantage of this approach is that it is easy to implement in a distributed environment; the main disadvantage is that some transactions may execute for longer than expected and be rolled back when they are not in fact deadlocked.

- *Graph-based*: This explicitly tracks waiting transaction dependencies by constructing a waits-for graph: nodes are waiting transactions and edges are waiting situations. The main advantage of this approach is that it is guaranteed to detect all deadlocks, whereas the main disadvantage is that in a distributed environment it can be costly to execute.

A slight variation on the timeout-based approach exists in some transaction systems, where timeouts can be associated with lock acquisition, such that the system will only block for the specified period of time. If the lock has not been acquired by the time this period elapses, it returns control to the application indicating that the lock has not been acquired. It is then up to the application to determine what to do; for example, it may be possible to obtain the required data elsewhere or to ask for a lock in a different mode. The advantage of this approach is that a transaction is not automatically rolled back if it cannot acquire a lock, possibly saving the application lots of valuable time; the disadvantage is that it requires additional effort on behalf of the application to resolve lock acquisition failures. However, for many objects and applications, this is precisely the best place to resolve lock conflicts because this is the only place where the semantic information exists to know what (if anything) can be done to resolve such conflicts.

Durability

The durability (or persistence) property means that any state changes that occur during the transaction must be saved in such a manner that a subsequent failure will not cause them to be lost. How these state changes are made persistent is typically dependent on the implementation of the transaction system and the resources that are ultimately used to commit the work done by the transactional objects. For example, the database will typically maintain its state on disk in order to ensure that a machine failure (e.g., loss of power) does not result in loss of data.

NOTE The durability property can never be an absolute guarantee, since a catastrophic failure (e.g., corruption of hard disk) could ultimately result in total loss of information.

Services and Participants

Up to this point we've been fairly general with the terms "participant" and "service." However, most transaction processing systems make a clear distinction between a service (also known as a transactional object) and the participants that are controlled by the transaction coordinator. This distinction is important because it shows where the responsibilities lie for the ACID properties:

- *Transactional object/service*: This is the object or application entity that encapsulates the business logic or work that is required to be conducted within the scope of a transaction. This work cannot be committed by the application unless the transaction also commits; control is ultimately removed from the application and placed into the transaction's domain. An example (shown in Figure 1-4) of such an object would be an object (e.g., EJB) that is responsible for buying a seat on a flight given user input; only if the user commits the transaction does the ticket for the flight actually get purchased. This means that the EJB cannot by itself make the work it is asked to do happen: that is down to the transaction and the associated transactional participant. However, it is the service that is ultimately responsible for *consistency* and *isolation*. It may do this with help from the resources it uses to accomplish the business work (e.g., databases typically provide concurrency control) or through an explicit concurrency control service.
- *Transactional participant/resource*: This is the entity that takes part in the two-phase commit protocol and controls the outcome of the work performed by the service. In the example above, and shown in Figure 1-5, if the flight purchasing EJB uses a database to store information on seat availability, it will typically access this information via a JDBC driver; SQL statements will be sent to the database for processing (e.g., how to reserve seat 12E) via the driver, but these statements will be tentative and only commit when (and if) the transaction does so. In order to do this, the driver/database will associate a two-phase commit participant (an `XAResource`, as we see in Chapter 2) with the transaction. When the transaction terminates (either commits or rolls back), it informs the `XAResource`, which will in turn inform the database of the transaction outcome, by committing or rolling back the state changes. Obviously the participant, working with the database in this case, must be able to identify the work that the transactional object did within the scope of a specific transaction and either commit the work or roll it back. The participant takes part in the two-phase commit protocol to ensure atomicity and is also responsible for durability (it must make state changes on behalf of the service provisionally durable during the prepare phase, for example).

So far, we've looked at the basics of ACID transactions and the protocols that go to make up the various guarantees. In the following section we extend our investigations into the two-phase commit protocol and illustrate various optimizations that may be available in certain implementations.

NOTE You should be aware that it is not necessary for a transaction system to implement any of these optimizations.

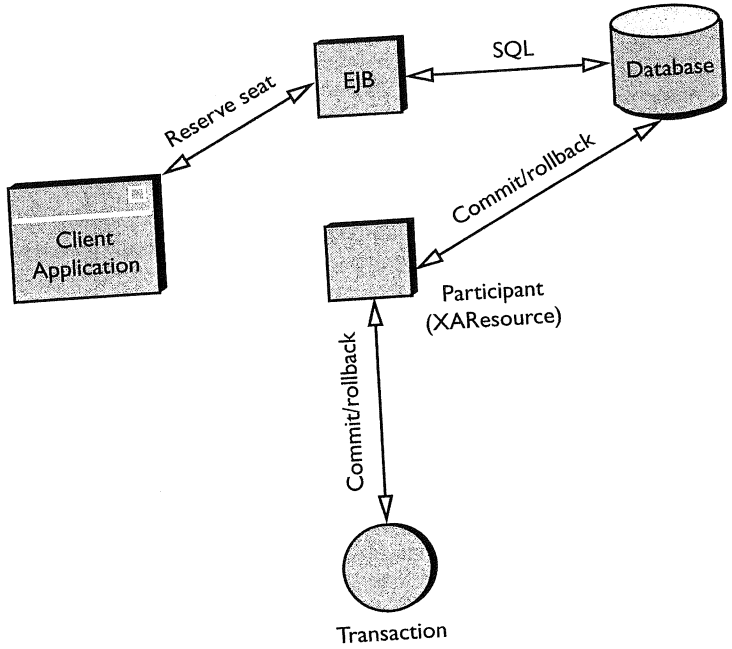

Figure 1-4 Transactional object and participant.

Two-Phase Commit Optimizations

As we've seen, the two-phase commit protocol is the standard mechanism for achieving atomic outcomes for transactions. When you use a transaction service, it is almost certain that it will use two-phase commit. To support high volume and performance applications, most transaction services will support optimizations to the protocol. There are several variants to the standard two-phase commit protocol that can have significant impact on performance and failure recovery. We shall briefly describe the three most common variants on the protocol that are found in most modern transaction processing systems:

- *Presumed abort*: If a transaction is going to roll back then it may simply record this information locally and tell all enlisted participants (even this record is actually optional). Failure to contact a participant has no affect on the transaction outcome; the coordinator is informing participants as a courtesy. Once all participants have been contacted (or as many as can be) the information about the transaction can be removed. If a subsequent request for the transaction's status occurs, there will be no information available and the requestor can assume that the transaction has aborted (rolled back). In some situations, the courtesy message from the transaction could be omitted entirely, requiring participants to eventually call back to the transaction to determine the outcome. (Although this can lead to an increased chance of heuristic outcomes, as we see later.) This optimization has the added benefit that no information about participants need be made persistent until the transaction has decided to commit (i.e., progressed to the end of the prepare phase), since any failure prior to this point will be assumed to be an abort of the transaction.
- *One-phase*: If there is only a single participant involved in the transaction, the coordinator need not drive it through the prepare phase—there is implicit consensus among participants in this case. Thus, the participant will simply be told to commit and (in some variants) the transaction coordinator need not record information about the decision since the outcome of the transaction is solely down to the participant.
- *Read-only:* When a participant is asked to prepare, it can indicate to the coordinator that it is responsible for an object/service that did not do any work during the course of the transaction, or at least did not do any work that modified any state. This is known as a read-only participant and it does not need to be informed about the outcome of the transaction since the fate of the participant has no affect on the transaction. As such, a read-only participant can be omitted from the second phase of the commit protocol.
- *Last resource commit:* In some cases it may be necessary to enlist participants that aren't two-phase commit aware into a two-phase commit transaction. As we've already seen with the one-phase optimization, if there is only a single resource then there is no need for two-phase commit. However, what if there are multiple resources in the transaction? In this case, the Last Resource Commit optimization comes into play. It is possible for a *single* resource that is one-phase aware (i.e., can only commit or roll back, with no prepare), to be enlisted in a transaction with two-phase commit aware resources. The coordinator treats the one-phase aware resource slightly differently, in that it executes the prepare phase on all other resource first, and if it then intends to commit the transaction it passes control to the one-phase aware resource. If it commits, then the coordinator logs the decision to commit and attempts to commit the other resources as well. As you can imagine, there are failure scenarios where loss of atomicity is possible here, but we go into more details in Chapter 2.

If we look at Figure 1-5, we see how the transaction coordinator interacts with the transaction log and the various transaction participants. In the figure, we assume an implementation of the presumed abort protocol and for simplicity we ignore the read-only optimization. As we can see, prior to the prepare phase, there is no need for the coordinator to interact with the log, since any failures of the application or transaction will simply cause the transaction to roll back. Then the coordinator enters the prepare phase and sends the prepare message to the participants. If any

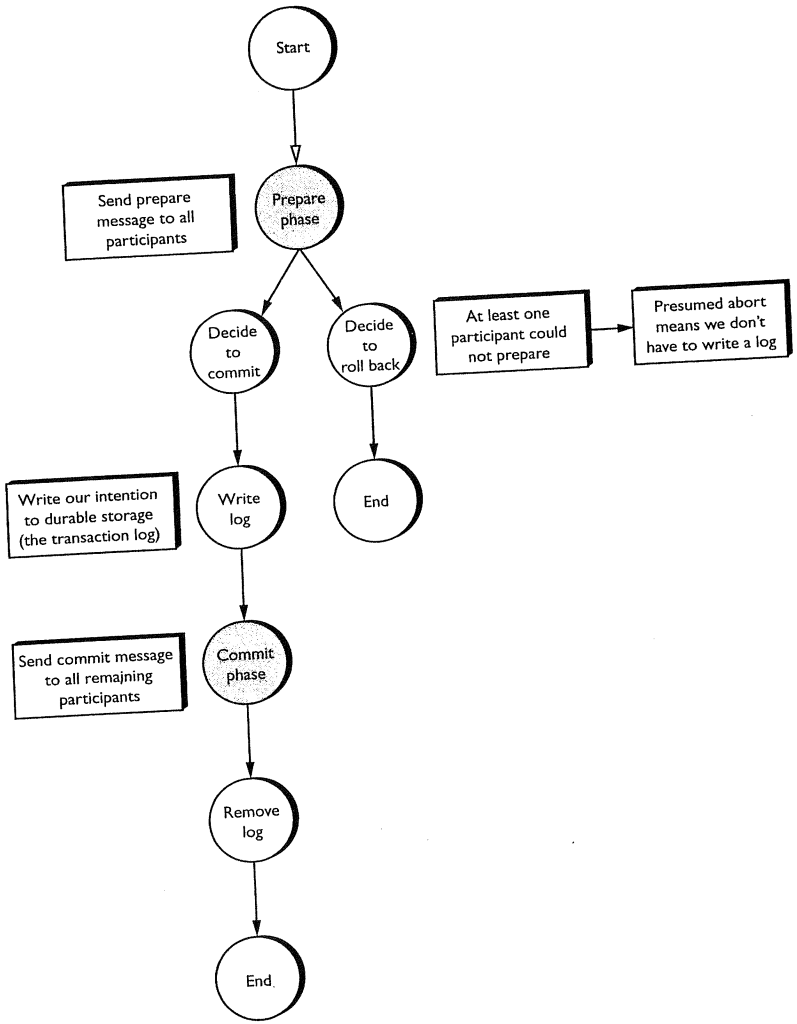

Figure 1-5 The coordinator, participant and transaction log interactions.

of the participants votes to roll back or fails to respond, the coordinator decides to roll back the entire transaction. As we saw earlier, there is no requirement in the presumed abort protocol for the coordinator implementation to write a log stipulating its intention to roll back.

However, if the prepare phase completes successfully and the coordinator decides to commit, then it must record this fact in the transaction log. (If the entire transaction used read-only participants, then no log updates will be required and the transaction can end.) Additional information such as references to the participants will also typically be recorded. At this point the coordinator can enter the commit phase and inform all participants that they should commit. Assuming this phase completes successfully, the transaction log can be removed.

If the commit phase does not complete in its entirety, the log will remain in existence, usually for failure recovery mechanisms to use later to try to finish the transaction. As we see later, it is possible for the second phase to fail in a number of ways. For example, the coordinator could crash, participants could find that they are unable to commit, network connections could fail, and so on. Luckily in most cases mature transaction systems have solutions to these problems that don't lead to data inconsistencies. Probably the worst type of failure during the second phase is that which leads to *heuristic outcomes*, but we deal with that in the next section.

So why is this important? Despite the fact that hardware gets faster year after year, it's still a fact that most durable store implementations (raw operating system file systems, databases, etc.) are the performance bottleneck for transaction services. Although durable storage media such as hard disks are faster than they were years ago, the performance gains of your typical processor have been much greater. What this means is that any attempt to reduce the number of disk accesses can also help to improve performance. Therefore, the fact that in a presumed abort protocol, the transaction coordinator need not access the durable log until it has decided to commit means that any potential bottleneck only occurs when required: when the transaction commits.

Earlier in this chapter we described the two-phase commit protocol that is used between coordinator and participants when terminating a transaction. In the following section we discuss another protocol that may also be used during termination and is yet another possible optimization available to transaction users.

Synchronizations

You may think that two-phase commit is sufficient for your transactional needs. However, if you want your applications to perform, you'll quickly be looking for ways to reduce the amount of time your application interacts with the persistence store. Caching of an object's state (e.g., an entire database table) in the local memory and operating on that cached state for the duration of a transaction can significantly improve performance over the alternative: continually going back and forth to the database. However, what happens if you update that cached state? You'll then need some way of forcing that state back to the original persistence store *prior* to the transaction committing.

In many transaction systems this is accomplished through *Synchronization participants*. Synchronizations are informed that a transaction is *about* to commit, so they can, for example, flush volatile (cached) state, which may have been used to improve performance of an application, to a recoverable object or database prior to the transaction committing. They are then informed when the transaction has completed and in what state it completed.

Let's take a look again at our banking example and try to show how Synchronizations could be used there. If you remember, the bank maintains the information on its accounts on a durable store, such as a database or disk. Now let's suppose that along with the amount of money held in each account, the bank maintains contact information on behalf of each client (name, address, home phone number, work phone number, and so on.) As part of its service to its clients, the bank supports the online update of this information by its clients, but this has to happen within a transaction.

Now, if Mr. Smith wants to update his contact details, a straightforward way of doing this would be for the bank to give direct access to the database on which the information resides. So, if he changes his address, that translates to a provisional update at the database; it has to be provisional, because Mr. Smith isn't done yet and so the transaction hasn't committed. If Mr. Smith wants to update a lot of information, this may take a while because each update (and even each read) requires access to the database.

Because of criticism of the poor performance of this service from its clients, the bank decides to re-implement slightly and use Synchronizations. When Mr. Smith first accesses his personal details, a cached copy of all of the information is (transparently) taken into the bank servers' memory and managed on behalf of the transaction by a Synchronization. Mr. Smith doesn't see any difference in terms of what he can do, but he will see a noticeable performance improvement. When Mr. Smith is eventually finished, the transaction terminates and this will cause (via the Synchronization) the cache to be flushed to disk before the two-phase commit protocol executes on the database.

Synchronizations essentially turn the two-phase commit protocol into a four-phase protocol:

- Before the transaction starts the two-phase commit, all registered Synchronizations are informed. What they do will obviously depend upon the specific implementation, but as we've seen, typically an implementation will flush a cached copy of some transactional object state to a database or other repository where its ultimate fate will be controlled by a participant in the two-phase commit protocol. Any failure at this point (the Synchronization fails to flush the state, for example) will cause the transaction to roll back. Note that synchronizations do not need to flush their state if the termination command for the transaction is abort: the changes would be rolled back.
- The coordinator then conducts the normal two-phase commit protocol. This protocol does not involve the Synchronizations.
- Once the transaction coordinator has finished the two-phase commit protocol, all registered Synchronizations are informed. However, this is a courtesy invocation because any failures at this stage are ignored: the transaction has terminated so there's nothing to affect.

Now that we've looked at possible optimizations to the traditional transaction protocols, let's consider how failures are tolerated and recovered from by transaction implementations.

We've already hinted at one of the most significant types of failure that can occur in a transaction system, so let's now take a detailed look at heuristic transactions. It's very important that we describe what heuristic transactions are, because they can mean a lot of work for you or a system administrator to resolve. Unfortunately, as we're about to see, they can be almost impossible to avoid entirely, especially in a distributed environment. Heuristic transactions are an inevitable fact of life.

Heuristic Transactions

In order to guarantee atomicity, the two-phase commit protocol is necessarily blocking. If the coordinator fails, for example, any prepared participants must remain in their prepared state until they hear the transaction outcome from the coordinator. When discussing failure recovery, we saw how recovery may be driven from both the top-down (coordinator to participant) and the bottom-up (participant to coordinator). However, in order for recovery to occur, machines and processes obviously need to recover! In addition, even if recovery does happen, the time it takes can be arbitrarily long.

What this means is that as a result of failures, participants may remain blocked for an indefinite period of time even if failure recovery mechanisms exist. Some applications and participants simply cannot tolerate this blocking (for similar reasons as deadlock). Imagine poor Mr. Smith having to wait days to be able to access his bank account again. As such, relatively early on in the development of transaction processing systems there was work on how to address this issue in a controlled manner.

To break the blocking nature of two-phase commit, participants that have gotten past the prepare phase are allowed to make autonomous decisions as to whether they commit or rollback: such a participant *must* durably record this decision in case it is eventually contacted to complete the original transaction. If the coordinator (or failure recovery mechanisms) eventually informs the participant of the transaction outcome and it is the same as the choice the participant made, there is no problem. However, if it is contrary, then a possibly non-atomic outcome has happened: a *heuristic outcome*, with a corresponding *heuristic decision*.

Just because a participant makes a heuristic choice does not automatically mean that the transaction outcome will be a heuristic. There are two levels of heuristic interaction:

- Participant-to-coordinator: If the participant makes an autonomous decision, it must assume that it has caused a heuristic and remember the choice it made so that eventually (and somehow) this could be resolved. At this point the participant is in a *heuristic state*.
- Coordinator-to-application: If the participant(s) really did make a choice contrary to the coordinator, then a heuristic outcome must be reported to the application. The transaction will still be marked as either committed or rolled back, but it will also be in a heuristic state, which will reflect the heuristic state(s) of the participant(s).

Some transaction service implementations will strive very hard to prevent heuristic outcomes from occurring. For example, let's assume the first phase of the termination protocol completes successfully and the coordinator decides to commit; if the very first participant the coordinator communicates with during the second phase tells the coordinator that it has rolled back, despite previously telling the coordinator it could commit, then obviously a heuristic could result. However, if the coordinator then changes its mind to match the participant's rollback decision and tells all other participants to roll back, no heuristic has actually happened: the state of all participants matches the coordinators (despite the fact that the coordinator actually changed its mind part way through the process!).

How a heuristic outcome is reported to the application and resolved is usually the domain of complex, manually driven system administration tools, since in order to attempt an automatic resolution requires semantic information about the nature of participants involved in the transaction, which is normally not available to most transaction processing systems. In Chapter 3, The Object Transaction Service, we see how the CORBA OTS specification maps heuristics to language-level exceptions that can be thrown by the transaction API. However, heuristics could occur after the transaction has officially terminated: for example, if not all participants can be contacted during the second phase of the commit protocol, the failure recovery system may be responsible for completing the transaction later. If that happens, any heuristic outcomes cannot be reported directly to the application, which may have long since terminated. It is for this reason that versions of the OTS allow heuristic reporting to be made through an external event-notification service.

Remember that participants in a heuristic state must remember the decision they took. Once the transaction has been resolved, these participants can be informed about the resolution and may then safely remove the information from durable storage.

Let's look the a specific example shown in Figure 1-6, where a transaction (shown by the Coordinator) is committing updates performed on two databases (shown by RDBMS). The two RDBMS participants confirm their ability to commit the transaction and obviously store this decision in a durable storage system. The resources used within the transaction (e.g., database tables) then remained blocked until they receive the final transaction outcome (are unavailable for other transactions/users).

The transaction moves to the second phase of the commit protocol and sends both participants the commit decision. Unfortunately, between the two calls the network connection between the coordinator and the second database fails. The first database has already committed (correctly), while the second does not receive the second phase message and hence keeps the resources blocked, pending the decision of the transaction manager. After some implementation specific period of time the second RDBMS (or the system's administrator) takes the autonomous decision to roll back the transaction in order to free these resources.

In this particular case, data integrity is compromised, as the entire transaction is no longer atomic. When the network connection is eventually restored and the coordinator's second phase message is eventually retransmitted (probably by the failure recovery system), the transaction

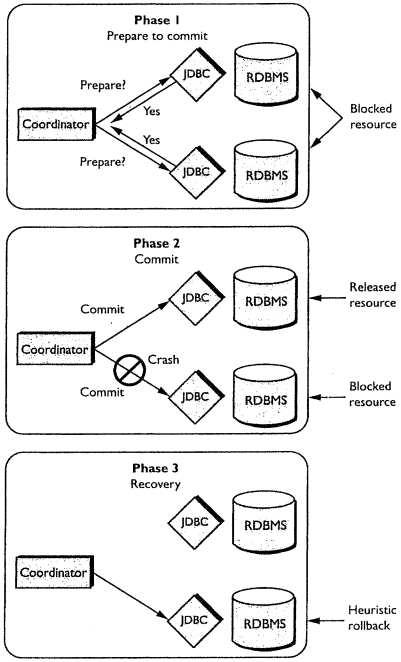

Figure 1-6 Example of heuristic outcomes.

system will determine that the second database took a decision that resulted in non-atomic behavior.

If the network failure had been between the coordinator and the *first* database, then the coordinator could have decided to roll back the transaction to try to ensure consistency. This would be based on the assumption that since no second phase message has been seen by any participant, the decision to commit hasn't been seen outside of the coordinator. In general this is probably the safest option to take in this situation, but it can't guarantee not to lead to heuristic outcomes. In this example, the first database may decide to autonomously commit if it doesn't receive the coordinator's second phase message.

When a participant makes a heuristic decision is obviously up to the participant: some may decide never to make an autonomous choice for fear of causing a non-atomic outcome. Others may choose when they have not heard from the coordinator within a long period of time after being asked to prepare, since most transaction commits take seconds or minutes to complete.

Obviously the choice the participant makes (to commit or to roll back) will depend upon the participant implementation and possibly the application/environment in which it finds itself.

The possible heuristic outcomes are:

- Heuristic rollback: the commit operation failed because the participants unilaterally rolled back the transaction.
- Heuristic commit: an attempted rollback operation failed because all of the participants unilaterally committed. This may happen if, for example, the coordinator was able to successfully prepare the transaction but then decided to roll it back (e.g., it could not update its log) but in the meantime the participants decided to commit.
- Heuristic mixed: some updates (participants) were committed while others were rolled back.
- Heuristic hazard: the disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

As we saw in the example, heuristic decisions should be used with care and only in exceptional circumstances since there is the possibility that the decision will differ from that determined by the transaction service and will thus lead to a loss of integrity in the system.

Heuristic outcomes are perhaps the worst type of failure that can befall an application using transactions. However, as we've seen, any good transaction system will provide support to help alleviate the problems that may arise from them. This support is usually part of a much larger subsystem component of the transaction system: the failure recovery system, which we discuss in more detail in the next section.

NOTE In some cases an application may not want to be informed of heuristic outcomes at all. (We would caution against this, for fairly obvious reasons.) In this case, when the application instructs the coordinator to terminate the transaction, it is possible for a transaction service implementation to return control to the application as soon as the first phase of the commit protocol has completed, spawning a separate (asynchronous) thread to perform the second phase (commit or rollback). This has obvious performance benefits, but failures in the second phase may not be made available directly to the application. Some commercial transaction services provide this option (e.g., the Arjuna Transaction Service), but it should be used with caution.

The Transaction Log

Although most users of transactions will see durability from their application's point of view, there is also an aspect of durability within the transaction system implementation itself. In order to guarantee atomicity in the presence of failures (for both the transaction coordinator and resources), it is necessary for the transaction service itself to maintain state. For example, in some implementations the coordinator must remember the point in the protocol it has reached (i.e., whether it is committing or aborting), the identity of all participants that are registered with the transaction and where they have reached in the protocol (e.g., whether they have received the prepare message). This is typically referred to as the *transaction log*, though this should not be

interpreted as implying a specific implementation. Some implementations may maintain a separate log (file) per transaction, with this information recorded within it and removed when it is no longer needed. Another possible implementation has a single log for all transactions and the transaction information is appended to the end of the log and pruned from the log when the respective transaction completes.

Let's look at what happens at the participant in terms of durability, when it's driven through the two-phase commit protocol, shown in Figure 1-7. We also use the term *original state* to mean the state the participant controls prior to the transaction starting, and *new state* to mean the state that exists when the transaction is about to commit.

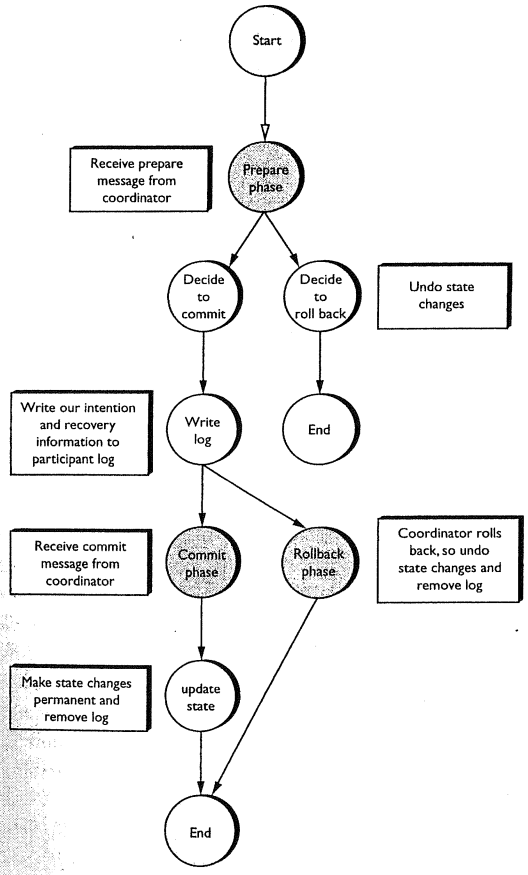

Figure 1-7 Participant log interactions.

As you can see, when the participant receives a prepare message from the coordinator it must decide whether it can commit or roll back. If it decides to roll back then it must undo any state changes that it may control (e.g., make sure that Mr. Smith's accounts are returned to their original state) and inform the coordinator; there is no requirement for durable access at this point. If the participant can commit, it must write its intentions to a durable store (*participant log*) along with sufficient information to either commit or roll back the state changes it controls. The format of this information is typically dependent on the type of participant, but may include the entire original and new states.

Once successfully recorded, the participant informs the coordinator that it can commit and awaits the coordinator's decision. When this second phase message arrives, the participant will either cancel all state changes (the coordinator wants to roll back), or if the coordinator wants to commit, make those state changes the current state (e.g., overwrite the original state with the new state). It can then delete the participant log and inform the coordinator.

Failure Recovery

Failures occur in all computing systems, both centralized and distributed. The more components that are involved with an application, the greater the chance of a failure occurring. As we mentioned earlier, in a distributed system failures are often independent—the failure of one component (process, machine, and so on.) does not necessarily cause the (immediate) failure of another.

Not only are transactions a good tool to use to ensure consistency in the presence of concurrent users or to obtain consensus between multiple participants/services, but they also offer an excellent fault-tolerance mechanism. Scoping work within a transaction, you immediately get the benefits of atomicity (all or nothing) despite failures: the transaction system must ensure that the work you do within the transaction either happens or appears to have never happened. All you will typically have to do is start and end the transaction.

However, failures can affect the transaction system itself. Although you may host your transaction system on more expensive (and hopefully more reliable) machines than your typical application, there can be no guarantees that those machines or the network connecting them to the rest of your environment won't fail. Enterprise-level transaction systems have usually had many man-years of effort put into their failure recovery and fault-tolerance capabilities. Without such capabilities, the benefits a transaction system offers are seriously downgraded. You will usually find that failure recovery is something that most open source or freeware transaction systems neglect entirely or simply don't consider to be important. Unfortunately, for a user it can often be the most important piece of the puzzle (Mr. Smith would be more than a little annoyed if the bank's transaction system lost his $400 due to a failure). It's something you don't want to discover that has been omitted until it is too late.

In order to cope with failures, transaction service implementations must possess some form of failure recovery subsystem. This subsystem ensures that results of a transaction are applied consistently to all resources affected by the transaction, even if any of the application

processes or the machine hosting them crash or lose network connectivity. In the case of a machine (system) crash or network failure, the recovery will obviously not take place until the system or network is restored.

Importantly, the original application does not need to be restarted in order to perform recovery: the responsibility for recovery is typically delegated to a separate recovery process, although this will depend strongly on the transaction service implementation being used.

Recovery after failure requires that information about the transaction and the resources involved survives the failure and is accessible afterward: this information (the transaction log mentioned previously) is held in some durable state-store and therefore available upon machine recovery. Typically the transaction log (or logs, depending upon the implementation) is scanned to determine whether there are transactions mentioned in it that require recovery to be performed. If there are, then the information within the log is used to recreate the transaction and the recovery subsystem will then continue to complete the transaction. What action the recovery subsystem performs will depend upon which flavor of two-phase commit the transaction system uses; for example, in a presumed abort protocol, the fact that a log entry exists implicitly means that the transaction was in the process of committing.

We have already described the ACID properties that using transactions provide. One obvious consequence of ACID is that until the recovery procedures are complete, resources affected by a transaction that was in progress at the time of the failure may be inaccessible. For database resources, this may be reported as tables or rows held by "in-doubt transactions." To do otherwise could result in inconsistencies in the application or data.

> **NOTE** What has been described so far assumes that recovery occurs from the transaction coordinator to the participant (top-down recovery). However, this is not necessarily always the case and in fact recovery may have to be driven from the participant to the transaction coordinator (bottom-up recovery).

So, for example, if a machine on which a participant resides fails and then recovers, the participant may need to enquire as to the status of the transaction. If participants waited for recovery to be driven from the coordinator then there are a number of issues that could cause problems. (This is not meant to be a complete list and is only presented to illustrate what a complex issue failure recovery can be and why many non-commercial implementations tend to undervalue it and underestimate how much effort is really required to do it justice.)

- If the coordinator has also failed and recovered, it may take some time before the recovery subsystem gets around recovering the specific transaction and hence resources may be inaccessible for longer than necessary.
- If a presumed abort protocol is used and the participant fails after having said it could prepare and the coordinator fails before it writes its log to say that it wants to commit, there will be no recovery on behalf of that transaction—no log entry for the transaction

means the transaction has rolled back as far as the recovery system is concerned. Hence, the participant will never get a termination message from the recovery subsystem.

Thus, most transaction systems require a failure recovery component to exist on both the coordinator and participant machines so that recovery can be driven in a bi-directional manner.

Implementing recovery is extremely complex and most public domain transaction service implementations (and some commercial products) don't provide it. So beware—find out about this feature before you invest time and money in using transactions. You don't want to discover that recovery is not supported after a critical failure has occurred. Without recovery mechanisms, atomic transactions may not be possible.

Let's consider our bank example and how a failure would affect it, as illustrated in Figure 1-8. When Mr. Smith begins to transfer his money he will (indirectly) start a transaction that will be used to manage the debit from the CurrentAccount and the credit to the SavingsAccount. Now let's assume that the debit/credit happens as per Mr. Smith's instructions. To do this, each

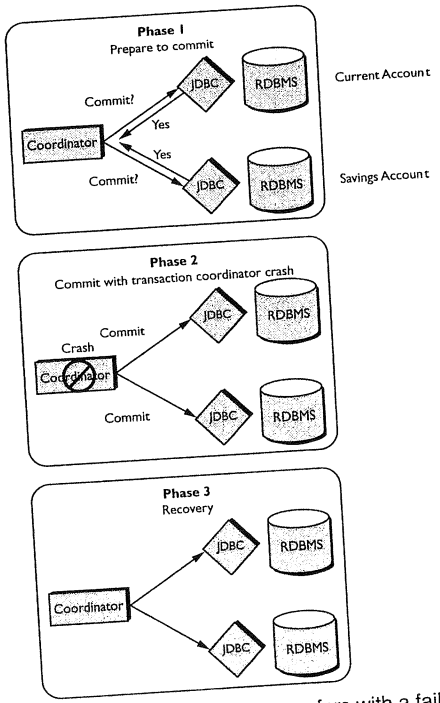

Figure 1-8 Bank account transfers with a failure.

of the bank account transactional services will have enlisted a participant with the transaction. When Mr. Smith informs the bank to perform the transaction, the bank will inform the transaction manager to commit. The transaction coordinator will execute the two-phase commit protocol among the registered participants. When it decides that it can commit, the transaction will write the log with information on the CurrentAccount and SavingsAccount participants; likewise, each of the two participants will previously have written enough information during the prepare phase to ensure that they can either commit or rollback the transfer should the coordinator inform them.

If a failure of the transaction coordinator occurs at this point (as shown in Figure 1-8) then the money (and the two bank accounts) will remain in a state of limbo. The money has not been transferred to the SavingsAccount but it also no longer resides in the CurrentAccount. Typically Mr. Smith won't be able to look at his accounts, because to do otherwise would break transaction semantics. However, because his bank uses an industrial strength transaction system, he can rest assured that his money hasn't been lost. Unfortunately, he'll just have to wait until the transaction coordinator machine recovers. When this eventually happens, the recovery subsystem will locate the transfer transaction in the log and essentially recreate the transaction. Once recreated, the transaction coordinator once again has the ability to complete the second phase of the two-phase commit protocol with the two account participants. At last Mr. Smith's money is transferred and he can check his accounts again.

Now let's consider a slightly more complex scenario, shown in Figure 1-9. During the post-prepare phase of Mr. Smith's transfer of money, a failure of the network stops the transaction coordinator from sending the commit message to the second database (the SavingsAccount); the CurrentAccount commits as instructed and Mr. Smith's $400 is removed. As we've already mentioned, the second database will have to wait for the second phase message before making a decision and the enlisted resources remain unavailable to other transactions. However, in this case the second phase message doesn't come quickly because of the network failure. So, after a period of time set by the administrator, the database makes a unilateral decision without waiting for the transaction coordinator; it decides to roll back the transaction updates (not accept to transfer of funds into the SavingsAccount) in order to free up the bank account (resources). This is obviously a heuristic decision, as the transaction has been committed and the $400 removed from the CurrentAccount.

Once the network connection is re-established, the crash recovery component will reconnect to the second database and will detect a heuristic exception, which will be transmitted to the administrator so that he/she can take the necessary compensatory action to bring the system back into a consistent state.

At this point we've covered many of the fundamentals of transaction processing, including optimizations and recovery from various failure scenarios. We've seen that transactions are an important tool that can be used to obtain fault-tolerance and ensure consistency in the presence of concurrent access to data. If you use transactions then you'll be placing your trust in these concepts as well as the specific service implementation. So, let's take a quick look at some trust implications that may not be immediately apparent, but which are important to understand.

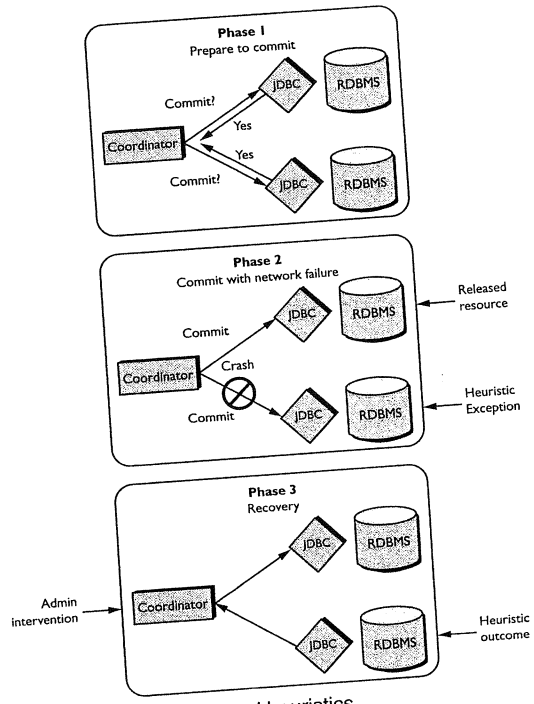

Figure 1-9 Recovery and heuristics.

Trust Implications

Something often overlooked by transaction implementers and users is that the two-phase commit protocol assumes a level of trust between the parties involved in the transaction. As we've seen, in any transaction there are essentially three actors:

- The participant that is driven through the two-phase commit protocol.
- The coordinator, with which participants register and that drives them through the commit protocol.
- The application logic that tells the coordinator whether it wants to commit or rollback the transaction.

Whatever the application logic might want (e.g., to commit) is never guaranteed by the coordinator, simply because failures can occur that force the transaction to roll back. In the earlier bank account example, if a transaction were used to control the transfer of money then the

work only actually happens when the transaction is committed (after step 4); if a failure occurs during the transfer then the transaction system will rollback, putting the money back into the CurrentAccount.

Assuming that the coordinator and participants behave correctly and no failures occur, then whatever the application logic tells the coordinator to do, the participants will follow. If Mr. Smith informs his bank to commit the transfer, then it will happen.

However, what happens if someone else manages to get control of the transaction? Suppose for example that someone else (Mr. Jones) is capable of intercepting Mr. Smith's directions to the bank and replaces them with his own. If this is the case, then they can obviously tell the bank to roll back even if Mr. Smith is happy with the transfer. This is an unlikely scenario, but it illustrates that the coordinator can only do what it is told to by its users. By itself a transaction coordinator does not possess semantic information about the environment in which it runs or its users; it can't tell the difference between Mr. Smith and Mr. Jones.

Now consider the case where one of the participants decides to act maliciously. By being involved in a transaction the participant has knowledge of the outcome of that transaction: knowing that a specific transaction has committed or rolled back could be important to some people (e.g., insider share trading, where knowing that a specific deal has just occurred could gain you an advantage on buying shares). Furthermore, the participant can affect that outcome through the two-phase commit protocol, e.g., always forcing it to roll back. In effect, a suitably implemented participant could cause a denial of service attack by always refusing to allow a transaction to commit.

In addition, the two-phase commit methods a participant must implement (prepare, commit and rollback) require a certain level of trust in order to work. For example, when told to prepare and then to roll back it is assumed by the coordinator (and ultimately by the user of the transaction) that a participant will undo the work it was told to. There is no guarantee though—it could lie. Suppose that during prepare an application sends to a participant a decryption key for a document and then calls rollback assuming that the participant will destroy the key. The participant may say it has but could in fact lie and keep the key for later.

What if the coordinator is illegal? Being the coordinator allows you to have final say on the actual outcome of the transaction (assuming all other entities involved in the transaction are not being malicious). Just because the application logic says to roll back does not mean that the coordinator actually has to roll back the participants—it could lie. Likewise, if the application logic says to commit and all participants say they can commit, the coordinator could still roll them back. Even more threatening (especially to application consistency) is the fact that the coordinator could tell some participants to commit and others to roll back.

We're not mentioning these "holes" in the protocol to worry you. Most industrial strength transaction systems have been around for years and often incorporate audit trail mechanisms to keep track of which entity did what and when. However, we feel that it is important for you to realize that just because you're using a "transaction system" does not mean that all of your troubles have been solved. As we shall see in later chapters, there is much more to an enterprise transaction system than "just" the two-phase commit protocol and we want you to be forewarned about the possible risks you might take by making that assumption.

So far we've only considered one type of transaction: the so-called ACID transaction. It may surprise you to know that there are quite a few different types of transaction models, each with its own advantages and disadvantages that may make it better suited to certain use cases. In the following section we look at some of the better-known transaction models.

Types of Transactions

The most common form of transaction that people will use is typically called a *top-level transaction*. It exhibits all of the ACID properties mentioned previously. However, traditional transaction processing systems are sufficient if an application function can be represented as a single top-level transaction. Frequently this is not the case. Top-level transactions are most suitably viewed as "short-lived" entities, performing stable state changes to the system; they are less well suited for structuring "long-lived" application functions (e.g., running for minutes, hours or days). Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources (e.g., locks) for a long time; further, if such a transaction aborts, much valuable work already performed could be undone. Several enhancements to the traditional flat-transaction model have been proposed and we shall briefly consider some of them in the following sections.

Nested Transactions

Given a system that provides transactions for certain operations, it is sometimes necessary to combine them to form another operation, which is also required to be a transaction. The resulting transaction's effects are a combination of the effects of the transactions from which it is composed. The transactions that are contained within the resulting transaction are *nested* (or *subtransactions*), and the resulting transaction is referred to as the *enclosing* transaction. The enclosing transaction is sometimes referred to as the *parent* of a nested (or *child*) transaction. A hierarchical transaction structure can thus result, with the root of the hierarchy being referred to as the *top-level transaction*, as shown in Figure 1-10, where each transaction is represented as an ellipse.

An important difference exists between nested and top-level transactions: the effect of a nested transaction is provisional upon the commit/roll back of its enclosing transaction(s), i.e., the effects will be recovered if the enclosing transaction aborts, even if the nested transaction has committed.

Subtransactions are a useful mechanism for two reasons:

1. *Fault-isolation*: If subtransaction rolls back (e.g., because an object it was using fails) then this does not require the enclosing transaction to roll back, thus undoing all of the work performed so far.
2. *Modularity*: If there is already a transaction associated with a call when a new transaction is begun, then the transaction will be nested within it. Therefore, a programmer who knows that an object requires transactions can use them within the object: if the

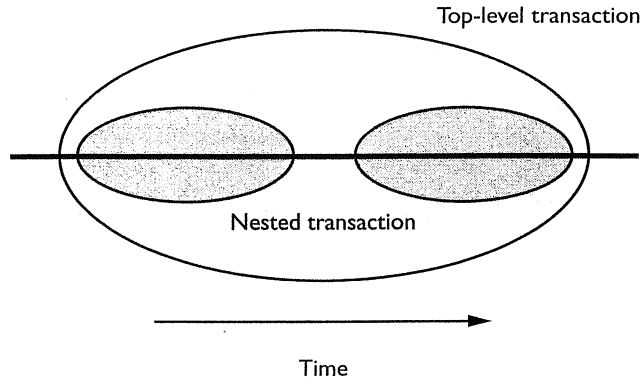

Figure 1-10 Nested transactions.

object's methods are invoked without a client transaction, then the object's transactions will simply be top-level; otherwise, they will be nested within the scope of the client's transactions. Likewise, a client need not know that the object is transactional, and can begin its own transaction.

> **NOTE** Because nested transactions do not make any state changes durable until the enclosing top-level transaction commits, there is no requirement for failure recovery mechanisms for them.

As far as isolation rules are concerned, locks that are acquired by a child transaction are inherited by the enclosing transaction when it commits. Typically those locks are then available for us in that transaction and any subsequently created children. However, there are other nesting models that restrict the availability of locks to only the parent, so that any newly created child transactions do not automatically have the same access rights on shared resources. This could mean that sibling transactions could conflict with one another.

Although nested transactions have been around in the literature for many years, they have not made much of an appearance in commercial transaction systems. There are a number of reasons for this and not least among them is that no commercial database supports subtransaction-aware participants. Since the majority of services use databases for persistence, this has always been a problem for the proponents of nested transactions. However, this is not to say that they aren't useful or that you can't find commercial transaction systems that support them. For example, the Arjuna Technologies Transaction Service supports nested transactions and provides nested transaction aware resources. The Encina transaction service available from IBM and IONA Technologies supports a limited form of nesting with X/Open XA transactions, by attempting to map them to top-level transactions.

One compromise that used to be common in some transaction systems was to allow a form of optimistic nesting when true nested transaction semantics were not available. Nested transactions could be started, but were grouped into the same top-level transaction such that nested commits would work fine, but a rollback in any nested transaction would require that the parent transaction would eventually be forced to roll back as well. This would allow things like workflow systems to start and end transactions, even if an underlying data store did not support nesting. As we see later in Chapter 3, the Object Transaction Service supports a similar model for its nested transactions.

Independent Top-Level Transactions

In addition to normal top-level and nested transactions, there are also such things as *independent top-level transactions* (also known as *nested top-level transactions*), which can be used to relax strict serializability in a controlled manner. With this mechanism it is possible to invoke a top-level transaction from within another transaction. An independent top-level transaction can be executed from anywhere within another transaction and behaves *exactly* like a normal top-level transaction, that is, its results are made permanent when it commits and will not be undone if any of the transactions within which it was originally nested roll back.

If the invoking transaction rolls back, this does not lead to the automatic rollback of the invoked transaction, which can commit or rollback independently of its invoker, and hence release resources it acquires. Such transactions could be invoked either synchronously or asynchronously. In the event that the invoking transaction rolls back compensation may be required.

Figure 1-11 shows a typical nesting of transactions, where transaction `B` is nested within transaction `A`. Although transaction `C` is logically nested within `B` (it had its `Begin` operation invoked while `B` was active), because it is an independent top-level transaction it will commit or rollback independently of the other transactions within the structure. Because of the nature of

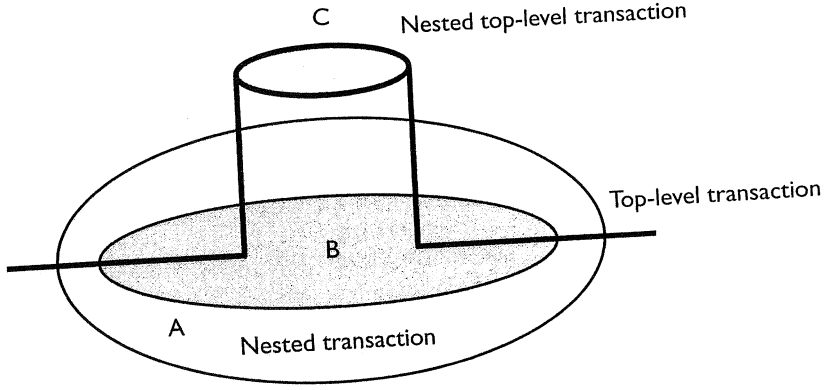

Figure 1-11 Independent top-level action.

independent top-level transactions they should be used with caution and only in situations where their use has been carefully examined.

Unlike nested transactions, nested top-level transactions can be found in the majority of commercial transaction systems. This is because you don't need any special support from your services or participants, i.e., there's no requirement for subtransaction-aware participants: ultimately a nested top-level transaction is a top-level transaction after all. And you might be surprised how useful they are and how relatively widely deployed nested top-level transactions are used.

Concurrent Transactions

Just as application programs can execute concurrently, so too can transactions (top-level or nested), i.e., they need not execute sequentially. So, a given application may be running many different transactions concurrently, some of which may be related by parent transactions, as shown in Figure 1-12.

We've already discussed the isolation rules for top-level and nested transactions. Whether those transactions are executed concurrently or serially does not affect the isolation rules: the overall affect of executing concurrent transactions must be the same as executing them in some serial order.

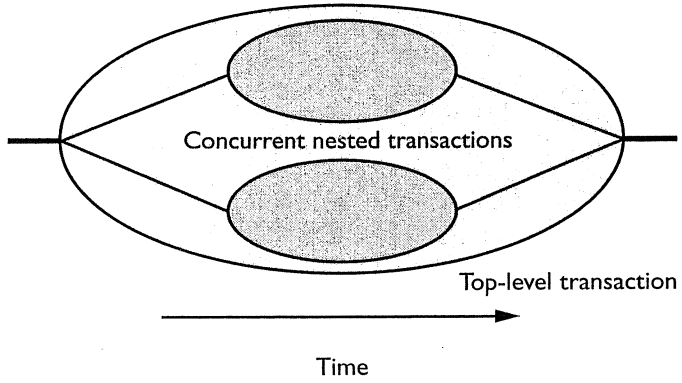

Figure 1-12 Concurrent transactions.

Glued Transactions

Top-level transactions can be structured as many independent, short-duration top-level transactions, to form a "logical" long-running transaction; the duration between the end of one transaction and the beginning of another is not perceivable and selective resources (e.g., locks on database tables) can be atomically passed from one transaction to the next, as shown in Figure 1-13. This structuring allows an activity to acquire and use resources for only the required duration of this long-running transactional activity. In the event of failures, to obtain transactional semantics for the entire long-running transaction may require compensation transactions that can perform forward or backward recovery.

As you might imagine, implementing and supporting glued transactions is not straightforward. Therefore, many (though not all) commercial transaction systems do not support them. However, as we shall see in Chapter 8, the advent of extended transaction mechanisms in general and specifically into J2EE may see that change.

So far we have assumed that transactions run in a local (non-distributed) environment. Although it is true to say that local transactions have their uses (e.g., updating a personal calendar), they really come into their own when used in a distributed manner. In the following section we'll describe those aspects of transactionality that are unique to distribution.

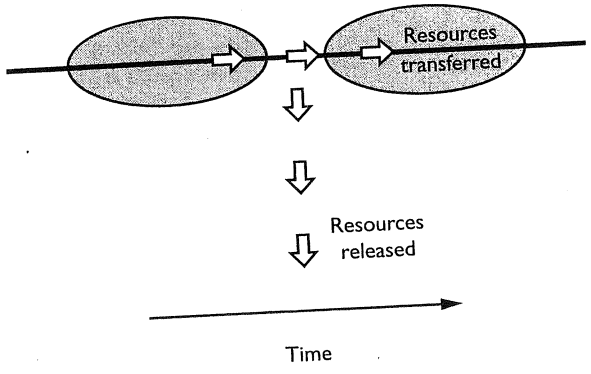

Figure 1-13 Glued transactions.

Distributed Transactions

In this section we look at what makes a transaction distributed. As you may imagine, as soon as you start to distribute transactions across a number of disparate machines (especially as the number of machines grows), many of the issues we've looked at (e.g., failure recovery) become much more complex. We're not going to be able to go into all of these details, but we touch on some of the more relevant aspects of distributed transactions.

The Transaction Context

In order for a transaction to span a distributed number of services/tasks, certain information has to flow between the sites or domains involved in the application. This is commonly referred to as the *context*. It is propagated in a distributed environment to provide a flow of information between remote execution environments, for example using the CORBA IIOP Service Context information. Typically this occurs transparently to the client and application services although in some cases the application programmer is able to explicitly propagate the context (e.g., as an extra parameter in a method call).

The context typically includes the following information:

- A transaction identifier that guarantees global uniqueness for an individual transaction.
- A coordinator location or endpoint address so participants can be registered by services.
- Implementation specific information, e.g., if the transaction system supports nesting of transactions, then this information may contain the transaction hierarchy that existed at the sending side in order that the importing domain may duplicate this hierarchy.

As shown in Figure 1-14, the context is propagated using whatever distribution mechanism is appropriate to the environment in which it is used (e.g., CORBA IIOP or SOAP). Many distribution infrastructures such as CORBA have a notion of message interceptors; as their name suggests, they intercept both incoming and outgoing messages and can add or remove information to/from the message. For example, at the application (client) an interceptor may be invoked prior to the remote service invocation going on to the network; when the message is taken from the network at the service-side, the corresponding interceptor is immediately invoked.

Transaction processing systems typically utilize interceptors to add the transaction context associated with the invoking thread to the outgoing application message. As shown in Figure 1-14, when the interceptor is called it asks the transaction service for the necessary information on the transaction (if any) associated with the application thread that is performing the remote service invocation. At the service side the transaction context is stripped off the message and associated with the thread that is about to do the work requested. As we see later, this may be an association with the original transaction resident at the client or may entail the domain importing the transaction context to create a local transaction proxy for the remote transaction. In either event, it appears as though the transaction seamlessly crossed process/machine boundaries.

The result of this is to produce what is typically called a transactional remote procedure call (TxRPC). A TxRPC has what is known as *exactly-once semantics*: if the remote call returns successfully it is guaranteed that it was executed once and only once. A basic, non-transactional RPC does not have this guarantee—such an RPC that fails may have been partially executed.

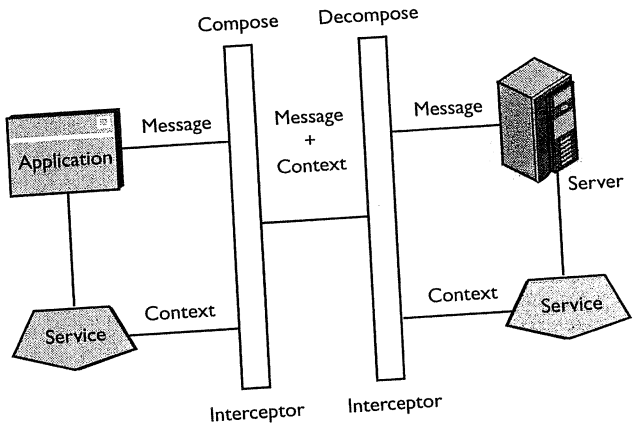

Figure 1-14 Services and context flow.

Interposition

Consider the situation depicted in Figure 1-15, where there is a transaction coordinator and three participants. For the sake of this example, let us assume that each of these participants is on a different machine to the transaction coordinator and each other. Therefore, each of the lines not only represents participation within the transaction, but also remote invocations from the transaction coordinator to the participants and vice versa:

- Enroll a participant in the transaction.
- Execute the two-phase commit protocol.

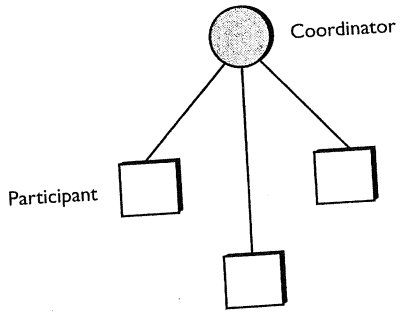

Figure 1-15 A distributed transaction.

In a distributed system there's always an overhead incurred when making remote invocations compared to making a purely local (within the same VM) invocation. Now the overhead involved in making these distributed invocations will depend upon a number of factors, including how congested the network is, the load on the respective machines, the number of transactions being executed, and so on. Some applications may be able to tolerate this overhead, whereas others may not. As the number of participants increase, so does the overhead for fairly obvious reasons.

A common approach to reduce this overhead is to realize that as far as a coordinator is concerned, it does not matter what the participant implementation does. For example, although one participant may interact with a database to commit the transaction, another may just as readily be responsible for interacting with a number of databases: essentially acting as a coordinator itself, as shown in Figure 1-16.

In this case, the participant is acting like a proxy for the transaction coordinator (the root coordinator): it is responsible for interacting with the two participants when it receives an invocation from the coordinator and collating their responses (and its own) for the coordinator. As far as the participants are concerned, a coordinator is invoking them, whereas as far as the root coordinator is concerned it only sees participants.

This technique of using proxy coordinators (or subordinate coordinators) is known as *interposition*. Each domain (machine) that imports a transaction context may create a subordinate coordinator that enrolls with the imported coordinator as though it were a participant. Any participants that are required to enroll in the transaction within this domain actually enroll with the subordinate coordinator. In a large distributed application, a tree of coordinators and participants may be created.

A subordinate coordinator must obviously execute the two-phase commit protocol on its enlisted participants. Thus, it must have its own transaction log and corresponding failure recov-

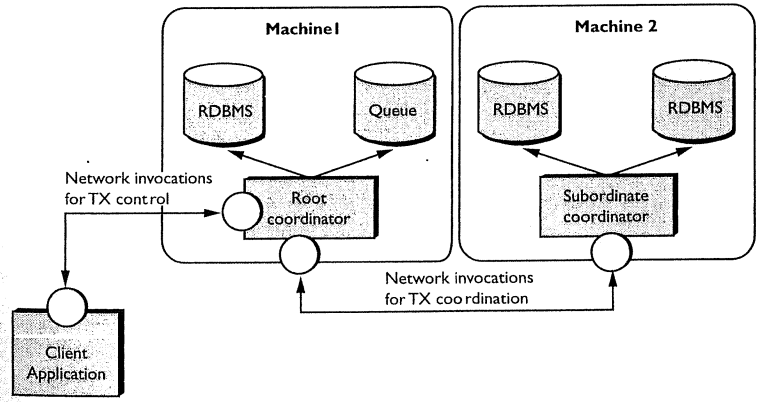

Figure 1-16 Participant coordinator.

ery subsystem. It must record sufficient recovery information for any work it may do as a participant *and* additional recovery information for its role as a coordinator. Therefore, it is impossible for a normal participant to simply be a subcoordinator because the roles are distinctly different; subcoordinators are tightly coupled with the transaction system.

> **NOTE** Not all transaction systems support interposition as many non-enterprise products see it as only being required for performance reasons. However, using interposed coordinators also allows an application to be broken down into administrative domains, where internal implementation details about participants are not exposed to other parts of the distributed system.

So the question then becomes when and why does interposition occur?

1. *Performance*: If a number of participants reside on the same node, or are located physically close to one another (e.g., reside in the same LAN domain) then it can improve performance for a remote coordinator to send a single message to a subcoordinator that is co-located with those participants and for that subcoordinator to disseminate the message locally, rather than for it to send each participant the same message.
2. *Security and trust*: A coordinator may not trust indirect participants and neither may indirect participants trust a remote coordinator. This makes direct registration impossible. Concentrating security and trust at coordinators can make it easier to reason about such issues in a large scale, loosely coupled environment.
3. *Connectivity*: Some participants may not have direct connectivity with a specific coordinator, requiring a level of indirection.
4. *Separation of concerns*: Many domains and services may simply not want to export (possibly sensitive) information about their implementations to the outside world.

So far we have talked about what transactions do and how they can benefit an application. So what exactly happens when your application starts a new transaction? In the next section we attempt to answer that question.

Controlling a Transaction

In order to talk about what happens when an application starts a transaction, we really need to talk in general terms since there are many different implementation choices that could affect what a specific transaction system does. Therefore, with this in mind, let's make a few initial assumptions:

- All transactions will be managed by a separate process (let's call it a Transaction Factory); most modern implementations—and most J2EE servers—provide optimizations that allow the factory to be co-located with users, but for the sake of our example it is easier to consider it as a separate entity. In J2EE terms this may be provided by the Java Transaction Service (JTS), which we describe in Chapter 3, but can be a proprietary implementation. So, although we might talk about JTS in the rest of this section, you shouldn't assume that it is a requirement; all transaction service implementations will have roughly equivalent terms and components.
- There is a transaction API that allows programmers to demarcate (start and terminate) transactions. This API will interface with the Transaction Factory. In J2EE terms this is the Java Transaction API (JTA), which we describe in Chapter 2.
- For each thread (unit of activity) in a process, the system maintains a mapping to the current transaction associated with that thread. When a thread creates a new transaction or terminates an existing transaction (e.g., via the API), this mapping is automatically updated. Luckily, if you use the JTA you won't need to worry about keeping track of which transaction is associated with which thread.
- All distributed invocations which occur within the scope of a transaction have the transaction context implicitly flowed between clients and servers so that remote invocations occur within the scope of the transaction that is active at the client. This is the domain of the underlying transaction service implementation; so, for example, a JTS implementation will use CORBA Portable Interceptors to augment the Service Context.

> **WARNING** We'd better stress that the above assumption is purely for the purposes of our example. Some systems don't necessarily propagate transaction information (context) on remove invocations even if there is a transaction associated with the client thread performing the invocation. You'd better check what your implementation does if this is a requirement or you could find that service requests don't happen transactionally.

Given these assumptions, the general architecture of the transaction system we use in this description is shown in Figure 1-17.

The commands offered by the API that allow an application to control a transaction are typically:

- *Begin*: start a new transaction.
- *Commit*: commit the transaction and make permanent any work performed within its scope. Obviously even though an application may wish for a transaction to commit, the coordinator and its participants will determine the final outcome.
- *Rollback*: roll the transaction back and undo any work performed within its scope. Sometimes this is also called *aborting the transaction.*

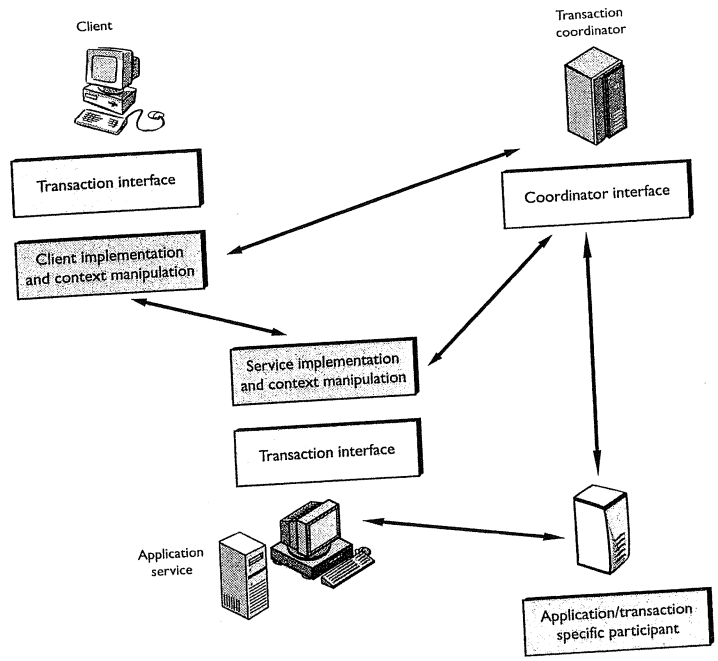

Figure 1-17 General transaction architecture.

As we've already mentioned, this control is typically provided to users through a suitable high-level API, such as the JTA or even the JTS (although as we see later, we would recommend against using the JTS raw as it offers a fairly low-level set of interfaces). However, there are some systems where no direct API exists and messages must be batched up by applications and sent to a transaction manager through a low-level API.

Using UML, Figure 1-18 illustrates what happens when the client issues *begin* on the transactional API (Tx API) to start a new top-level transaction. The Tx API first locates the Transaction Factory and issues a *create* request on it, which causes the factory to start a new transaction. Assuming the transaction is successfully started, the factory returns a reference to the client such that the client can use it for future requests and embed it within the transaction context.

When the transaction reference is returned to the client, the Tx API changes the invoking client thread's notion of what the current transaction is to this new transaction. If the system supported nested transactions, then the thread-to-transaction association would typically occur through a stack, where the new transaction was pushed onto the stack of transactions currently associated with the thread.

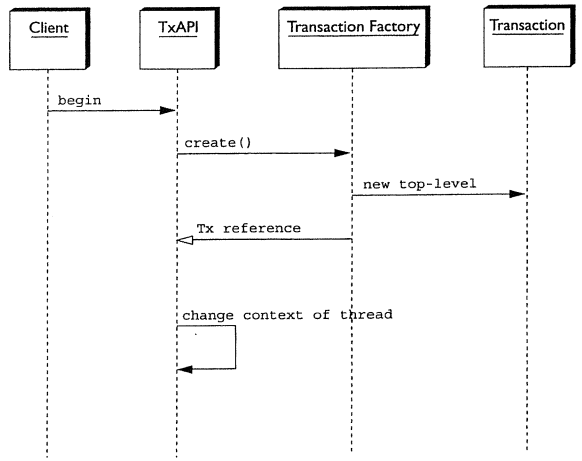

Figure 1-18 Starting a new top-level transaction.

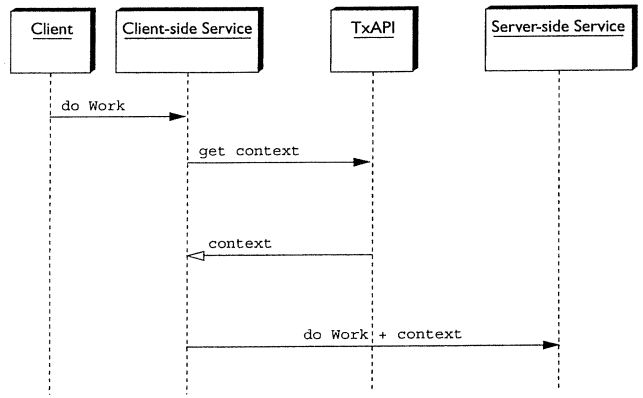

Figure 1-19 Remote transactional invocation.

Figure 1-19 illustrates the work involved when a remote invocation occurs within the scope of the transaction we have just created. Most distributed object models use client-side (local) proxies for remote services to make the fact of distribution transparent: both local and remote objects share the same interfaces and the client application interacts with the proxy which is then responsible for making the remote call. In this example, the client invokes the *doWork* method on the local proxy and its implementation is then required to determine if this invocation is occurring within the scope of an active transaction. Since it is, the reference to the

transaction is obtained and the context is then sent to the remote (server) object. Not shown is that at the server, the context is stripped from the invocation and associated with the server-side thread that does the real work, i.e., the inverse occurs.

NOTE How this actually occurs will depend upon the system used for distributed objects. For example, in CORBA, the IIOP Service Context can be enhanced using Portable Interceptor technology.

When transaction processing systems first began there was no such thing as a multi-threaded application as you may know them today. Having multiple "threads of control" execute within a single operating-system process was often difficult to obtain. In the following section we'll examine what it means to mix transactions and threads.

Transactions and Threads

In this section we consider some of the issues involved when using transactions in a multi-threaded environment. Although Java is inherently a multi-threaded language, it shouldn't come as a surprise to learn that transactions have been around for a lot longer than threads, so when the two came together there were some subtle interactions.

Checked Transactions

When transaction systems were first developed they were *single-threaded* (where a *thread* is defined to be an entity which performs work, e.g., a lightweight process, or an execution context.) Executing multiple threads within a single process was a novelty. In such an environment the thread terminating the transaction is, by definition, the thread that performed the work. Therefore, the termination of a transaction is implicitly synchronized with the completion of the transactional work: there can be no outstanding work still going on when the transaction starts to finish.

With the increased availability of both software and hardware multi-threading, transaction services are now being required to allow multiple threads to be active within a transaction (although it's still not mandated anywhere, so if this is something you want then you may still have to look around the various implementations). In such systems it is important to guarantee that all of these threads have completed when a transaction is terminated, otherwise some work may not be performed transactionally.

WARNING We can't ignore the fact that the transaction implementation must obviously be multi-thread safe!

Although protocols exist for enforcing thread and transaction synchronization in local and distributed environments (commonly referred to as *checked transactions*), they assume that communication between threads is synchronous (e.g., via remote procedure call). A

thread making a synchronous call will block until the call returns, signifying that any threads created have terminated.

However, a range of distributed applications exists (and yours may be one of them) which require extensive use of concurrency in order to meet real-time performance requirements and utilize asynchronous message passing for communication. In such environments it is difficult to guarantee synchronization between threads, since the application may not communicate the completion of work to a sender, as is done implicitly with synchronous invocations.

As we've just seen, applications that do not create new threads and only use synchronous invocations within transactions implicitly exhibit checked behavior. That is, it is guaranteed that whenever the transaction ends there can be no thread active within the transaction which has not completed its processing. This is illustrated in Figure 1-20, in which vertical lines indicate the execution of object methods, horizontal lines message exchange, and the boxes represent objects. Figure 1-20 illustrates a client who starts a transaction by invoking a synchronous "begin" upon a transaction manager. The client later performs a synchronous invocation upon object *a* that in turn invokes object *b*. Each of these objects is registered as being involved in the transaction with the manager. Whenever the client invokes the transaction "end" upon the manager, the manager is then able to enter into the commit protocol (of which only the final phase is shown here) with the registered objects before returning control to the client.

However, when asynchronous invocation is allowed, explicit synchronization is required between threads and transactions in order to guarantee checked (safe) behavior. Figure 1-21 illustrates the possible consequences of using asynchronous invocation without such synchronization. In this example a client starts a transaction and then invokes an asynchronous operation upon object *a* that registers itself within the transaction as before. *a* then invokes an asynchronous operation upon object *b*. Now, depending upon the order in which the threads are sched-

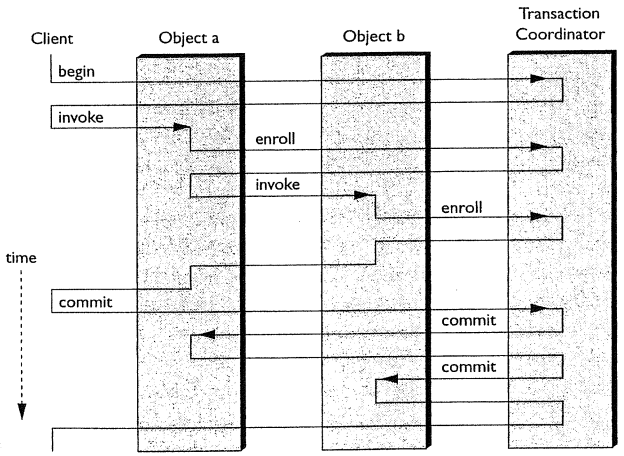

Figure 1-20 Synchronous checked transactions.

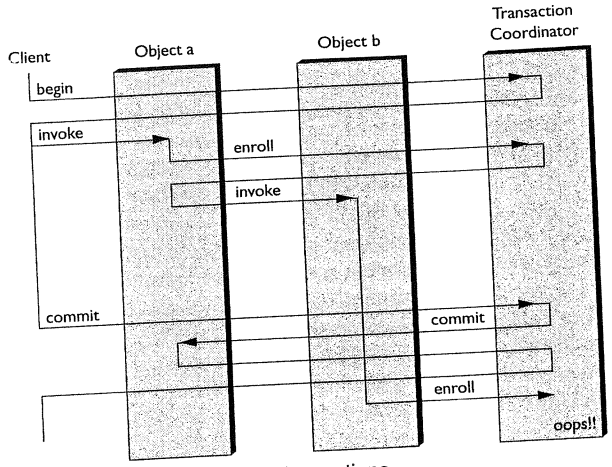

Figure 1-21 Asynchronous invocations.

uled, it's possible that the client might call for the transaction to terminate. At this point the transaction coordinator knows only of *a*'s involvement within the transaction so it enters into the commit protocol, with *a* committing as a consequence. Then *b* attempts to register itself within the transaction, and is unable to do so. If the application intended the work performed by the invocations upon *a* and *b* to be performed within the same transaction, this may result in application-level inconsistencies. This is what checked transactions are supposed to prevent.

Some transaction service implementations will enforce checked behavior for the transactions they support, to provide an extra level of transaction integrity. The purpose of the checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. A checked transaction service guarantees that commit will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. If the transaction is rolled back then a check is not required, since all outstanding transactional activities will eventually rollback if they are not told to commit.

As a result, most (though not all) modern transaction systems provide automatic mechanisms for imposing checked transactions on both synchronous and asynchronous invocations. In essence, transactions must keep track of the threads and invocations (both synchronous and asynchronous) that are executing within them and whenever a transaction is terminated, the system must ensure that all active invocations return before the termination can occur and that all active threads are informed of the termination. This may sound simple, but believe us when we say that it isn't.

Unfortunately, this is another aspect of transaction processing that many implementations ignore. As with things like interposition (for performance) and failure recovery, it can be an essential aspect that you really cannot do without in your application. Fortunately, J2EE places sever limitations on transactions and concurrency that result in checked transaction semantics on many of the typical ways in which transactions are used.

Not providing checked transactions is different from allowing checking to be disabled, which most commercial implementations support. In this case you typically have the ability to turn checked transactions off when you know it is safe (e.g., in a read-only application), to help improve performance. If you think there is the slightest possibility you'll be using multiple threads within the scope of a single transaction or may make asynchronous transactional requests, then you'd better find out whether your transaction implementation it up to the job.

Suspending and Resuming Transactionality

As we have seen, when a thread creates a transaction, typically that thread will then become associated with the transaction, i.e., if the thread subsequently asks, "what is my current transaction?" it will be returned as a reference to the one it has just created. As mentioned earlier, this is a bi-directional relationship, since the transaction needs to know which threads are running in it in order to ensure checked behavior.

In some applications, however, there may be a requirement for the thread to be disassociated from the transaction. For example, an application may require periods of non-transactional work for its threads, e.g., an update of a database that is not required to be undone should the enclosing transaction roll back. For that reason, most transaction systems provide a means whereby a thread can *suspend* its transaction association and later (or perhaps never) *resume* the association. This ability to modify a thread's transaction association can also be used to enable multiple threads to execute within the context of the same transaction and not just the creator.

Some distributed applications are what are often termed *loosely coupled*: their constituent components are often independent and execute over arbitrary lengths of time, with invocations often being made asynchronously. A client making an invocation on a service in such an application may not expect to get a response back for minutes, hours or days. Traditional top-level transaction semantics don't work efficiently in these applications. In the next section we examine how the introduction of queues (such as those found in JMS implementations) can solve this problem.

Transactions and Queues

. So far we have discussed what can be considered *direct transaction management*: the application synchronously starts the transaction, does work within its scope and then terminates it. This is very much in line with the RPC model that most distributed object systems support. However, there are some applications where this mode of operation is not suitable. For example:

- Transactional activities that may take a (subjectively) long time to complete and would block the client until finished.
- If the server fails or communication problems prevent the client from interacting with the server, the client may have to block waiting for the server to become available, or the application must be written to resubmit the request later.

- What if the client fails after having successfully submitted the request for work to be performed to the server? After executing the request, the server will be unable to deliver the response and the transaction will fail. A recovering client will not know whether the server failed before executing the request, or whether the response was simply lost.
- What if there is a pool of servers all offering the same (popular) service? If all clients use the same server, then it will quickly become overloaded and response times may drop. However, what is the best load-balancing strategy to use? If the application randomly distributes requests across these servers, there is no guarantee that the load will be evenly shared. What if a particular request is high priority and should be executed above all others?

Using an indirect transaction management facility based on queues or buffers of requests and responses can solve these problems. Instead of a client sending requests directly to services, they are sent to a *transactional queue*, as shown in Figure 1-22, which may order them based on priorities set by the queue, the application, the client, and so on. Adding (inserting) a request on a queue is performed within the scope of a transaction and the insertion only actually happens if the transaction commits. Therefore, if the client fails before the transaction commits, it knows that the request has not been performed.

Because the queue is transactional, it maintains persistent state so that it too can tolerate failures. How this state is maintained and what the configuration of the queue is will depend upon the queue implementation. For example, some implementations may store individual requests as separate persistent state entries, whereas others may store all request entries within a single persistent state. There are trade-offs to be made in terms of performance and concurrency control.

Depending upon the system configuration, an arbitrary number of clients may be able to insert an entry within the queue (essentially connected to the *request-insertion end*). Likewise, an arbitrary number of services may be able to remove requests from the queue (essentially connected to the *request-removal end*). It's important to realize that the clients and services connected to a queue need not be identical

As with the client and shown in Figure 1-23, requests are removed from the queue within the scope of a transaction that may span the execution of the work by the service. Thus, only if the request successfully executes the request will it be permanently removed from the queue: the failure of a server either as it removes the request or as it executes it will cause the dequeue transaction to roll back and (appear to) re-insert the request, making it available for another service to dequeue.

But what happens when the service has successfully completed (and removed) the request, i.e., what happens to the response? In just the same way as with requests may be inserted to a transactional queue, so can responses. It will typically be implementation or deployment specific as to whether a different queue is used for the responses than for the requests. Most implementations will also by default tie the enqueue of the response with dequeuing of the request. Other-

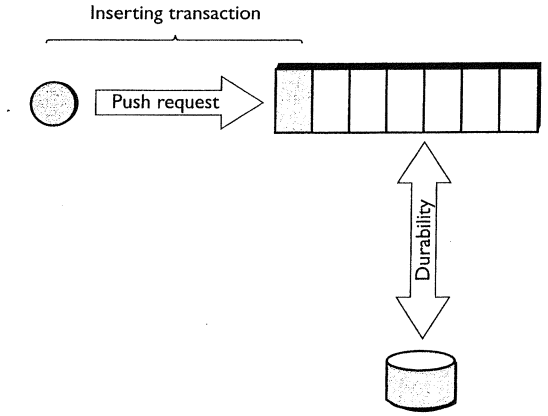

Figure 1-22 Transactional request enqueue.

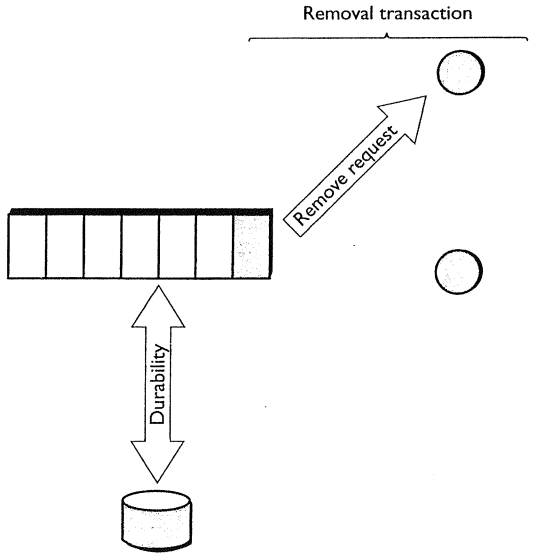

Figure 1-23 Transactional request dequeue.

wise, if two separate transactions are used a failure of the service could happen just after the request has been removed from the queue and before the response has been enqueued. This is illustrated in Figure 1-24.

Transactions ensure that only consistent state changes take place despite concurrent access and failures. However, they may be insufficient to ensure that an application makes forward progress, e.g., a transaction that rolls back because of a machine failure and then restarts may find that the machine has not yet recovered or fails again. It is possible to improve the probability that forward progress can be made by increasing the availability of an application's objects (resources) by replicating them on several machines and managing them through an appropriate replica-consistency protocol. The failure of a subset of these replicas may then be masked, allowing the application to continue. In the following section we give an overview of how replication and transaction techniques can combine to provide data consistency and forward progress in the presence of a finite number of failures.

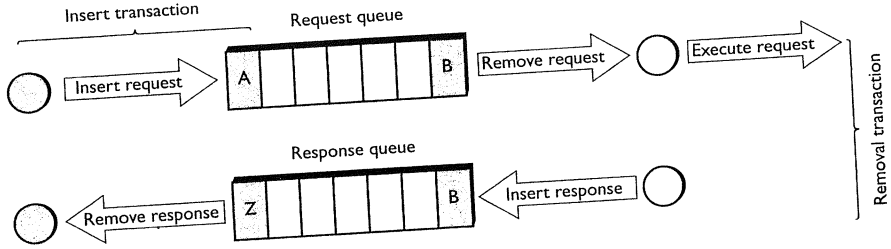

Figure 1-24 Request and response queues.

Transactions and Replication

What we're about to describe in this section is essentially concerned with replication of application-level objects and very few transaction systems (even commercial products) provide this kind of facility. However, as distributed systems grow in scale and complexity it is becoming more of an issue to improve availability for transactional applications and we believe that this sort of functionality will eventually become a critical requirement: you may already have the need today.

It is possible to construct fault-tolerant services (i.e., services capable of tolerating component failures) by replicating vital system components (both in software as well as hardware) and providing the notion of an *abstract component/service* to the users (one which exhibits the properties of a single component but is actually made up of many components). For simplicity we'll assume that replicated resources will reside on distinct nodes (workstations, or processors) in a distributed system (replicating on the same node has limited value, especially in terms of

improving *availability* in the presence of node failures). Replication can be used for two main reasons in a distributed system: increased availability, and increased performance. It's also possible that replication can also be used to provide a means of tolerating software design failures.

The management of replicated objects is a complex operation. The main difficulty arises from the fact that an object is not just data, but data (*instance variables*) plus code (*methods* or *operations* which operate on the instance variables); furthermore, method executions can result in calls on other objects. Thus the problem of managing replicated objects really amounts to that of managing *replicated computations*.

This problem can be best formulated in terms of the management of object groups (where each group will represent a replicated object), which are interacting via messages. To avoid any consistency problems it is necessary to ensure that a group appears to behave like a single entity in the presence of concurrent invocations and failures. If not managed properly, concurrent invocations could be serviced in different order by the members of a group, with the consequence that the states of replicas could diverge from each other. Group membership changes (caused by events such as replica failures and insertion of new replicas) can also cause problems if these events are observed in differing order by the users of the group.

Maintaining Information on Persistent Objects

To be able to use an object in a distributed environment, an application must obviously have a reference to it. If that object is replicated, then the information must be managed to maintain consistency. In this section we look at a *typical* way in which a system may identify and access persistent objects in a distributed system. It's important to realize that different implementations of the architecture we describe are possible, although the essential functionality will be the same.

As shown in Figure 1-25, clients typically request this information from a naming service by presenting it the name of the object (message 1). The service maps this name to a previously registered reference to the object and the application then directs its invocations to the object (message 2). In order for a client to make use of a persistent object it is necessary for there to be at least one machine (say α) which is capable of running a server for that object (i.e., it has access to code for the object's methods).

The server for a persistent object and its state need not reside on the same machine. Therefore, there must be at least one machine (say β) whose datastore contains the persistent state of the object. In order to load/store its state (message 4), α may have to contact the naming service to obtain a reference to that node (message 3).

Given the above description, we can see that in a replicated environment a naming service must maintain two sets of data:

1. Sv_A: for an object A, this set contains the names of nodes each capable of running a server for A.
2. St_A: this set contains the names of nodes whose object stores contain states of A.

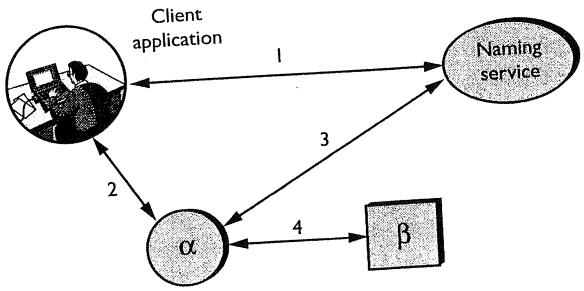

Figure 1-25 Object binding.

An object can become unavailable if all the nodes $\in Sv_A$ are down and/or all the nodes $\in St_A$ are down. In a non-replicated system, a common activation scheme for persistent objects activates an object at a fixed application server that also maintains the state. Therefore, a name server that maps an object name to the server location is all that is required. Exploitation of redundancy in the form of multiple application servers capable of hosting an object and multiple datastores capable of storing the state of an object is possible only after careful enhancements to the system.

Replication Protocols

We will consider the case of *strong consistency* which requires that the states of all replicas that are regarded as *available* be mutually consistent (so the persistent states of all available replicas are required to be identical). Object replicas must therefore be managed through appropriate replica-consistency protocols to ensure strong consistency. To tolerate K replica failures, in a non-partitionable network, it is necessary to maintain at least K+1 replicas of an object, whereas in a partitionable network, a minimum of 2K+1 replicas are necessary to maintain availability in the partition with access to the *majority* of the replicas (the object becomes unavailable in all of the other partitions). There are basically two classes of replication protocols:

- *Active replication*: more than one copy of an object is activated on distinct nodes and all copies perform processing, as shown in Figure 1-26. Active replication is often the preferred choice for supporting high availability of services where masking of replica failures with minimum time penalty is considered highly desirable. Since every functioning replica performs processing, active replication requires that they receive identical invocations in an identical order, and that the computation performed by each replica is deterministic.
- *Passive replication*: only a single copy (the primary) is activated, as shown in Figure 1-27; the primary regularly checkpoints its state to the object stores where states are

stored. One of the advantages of this form of replication is that it can be implemented without recourse to any multicast communication and does not require deterministic replicas; however, its performance in the presence of primary failures can be poorer than under no failures, because the time taken to switch over to a secondary is non-negligible.

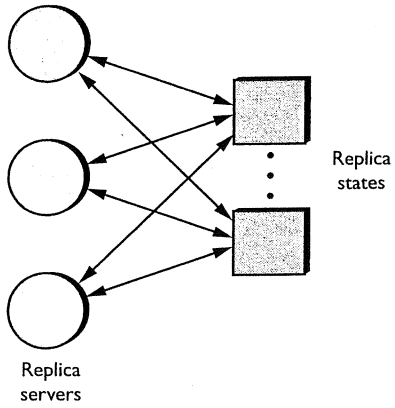

Figure 1-26 Active replication.

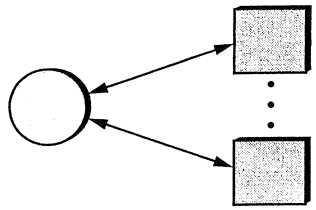

Figure 1-27 Passive replication.

Most replication protocols use replica group technology: A *group* is defined as a collection of distributed entities (objects, processes) in which a member can communicate with other members by multicasting to the full membership of the group. A desirable property is that a given multicast be *atomic*: a message is delivered to either all or none of the functioning members. An additional useful property is guaranteeing *total order*: messages are delivered in the same order to all the functioning members. These properties are ideal for replicated data management: each member manages a copy of data, and given atomic delivery and total order, it can be ensured that copies of data do not diverge.

There are three aspects of managing the replication of persistent objects that are common to both types of replication protocol:

1. *Object binding*: It is necessary to ensure that when an application activates a currently passive replicated object, it only uses those replicas that are (a) mutually consistent, and (b) contain the latest state of the object. If the object has been activated already, clients must be bound to the same replicas.
2. *Object activation and access*: A passive object must be activated according to a given replication policy. Part of object activation means locating the object state(s) and loading it/them into the correct object instance(s). Activated copies of replicas must be treated as a single group by the application in a manner that preserves mutual consistency. Communication between groups requires reliable distribution and ordering guarantees not associated with non-replicated systems: reliability ensures that all correctly functioning members of a group receive messages intended for that group and ordering ensures that these messages are received in an identical order at each member.
3. *Object passivation*: Once an application has finished using a replicated object, it is necessary to ensure that (a) new states of mutually consistent replicas get recorded to their object stores, and (b) information maintained by the system about which replicas are up-to-date remains accurate.

Integrating Replication and Transactions

The functionality provided by transactions and replication protocols overlap, requiring careful design of systems that wish to employ both techniques to avoid incurring unnecessary overheads. For example, many replication implementations rely upon reliable multicast communication to deliver invocations to each replica in the same order. However, the isolation property of transactions also imposes ordering guarantees (e.g., when concurrent clients attempt to perform conflicting invocations), and allows arbitrary interleaving of non-conflicting operations (e.g., concurrent read requests, or operations on different objects). Transactions impose ordering at the application level only when required. Multicast communication protocols typically enforce ordering at a lower level, where application specific knowledge is unavailable, and hence may consume system resources unnecessarily, e.g., ordering non-conflicting messages.

If you consider the three aspects of managing replicated objects mentioned above, then the typical way in which transactions and replication are integrated can best be illustrated with Figure 1-28. The shaded ellipses represent transactions that perform the binding or passivation phases and hence operate on the naming service information St_A and Sv_A mentioned previously. In the binding transaction, the naming service is contacted to either obtain a reference to the currently active object instances to use or, if the replicated object is not being used, those passive instances that can be used. Once the desired number of replicas has been activated, the transaction updates the naming service to inform subsequent clients.

The unshaded ellipse represents the application transaction that performs the real work on the replicated object, using whatever replica consistency protocol is deemed appropriate. As we mentioned earlier, this will typically involve multicast communication to ensure that all replicas get the same set of messages. Depending upon the replication protocol in use, the transaction may be able to commit in the presence of replica failures. If this is so, then when the transaction commits any replica failures must be recorded in order to prevent future users from using potentially out-of-date copies of the object. This is the job of the passivation transaction: it updates the St_A and Sv_A information.

So, we've seen how transactions and replication can be integrated together to provide consistency and forward progress. Although systems that conform to this style do exist, they are typically in the minority. In the next section we look at what you tend to find in terms of transactional high-availability offerings and also give an illustration of one system that does provide the kind of integration we've described.

Availability Measures in Current Application Servers

Commercial application servers tend to make use of multiple applications servers deployed over a cluster of machines in order to distribute load and improve availability in the presence of failures. In the case where specialist router hardware is used to mask server failures, all application servers are typically deployed over a cluster of machines. For example, a locally

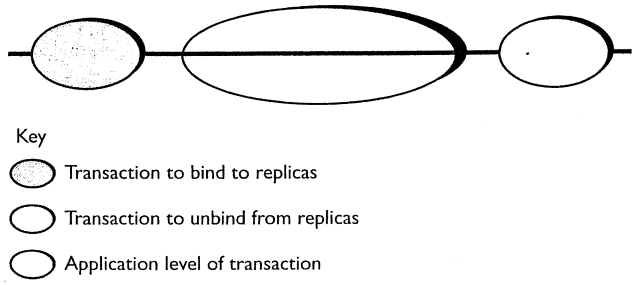

Figure 1-28 Transactions and replication scenario.

distributed cluster of machines with the illusion of a single IP address and capable of working together to host a Web site provides a practical way of scaling up processing power and sharing load at a given site.

Some commercially available application server clusters rely on specially designed gateway routers to distribute the load using a mechanism known as network address translation (NAT). The mechanism operates by editing the IP headers of packets so as to change the destination address before the IP to host address translation is performed. Similarly, return packets are edited to change their source IP address. Such translations can be performed on a per session basis so that all IP packets corresponding to a particular session are consistently redirected.

Alternately, load distribution and high availability can also be performed using a process group communication system as first supported by the ISIS system from Cornell University; for example, the JBoss open source application server has such a mechanism, called JGroups. There is a range of such group communication protocols (often termed *virtually synchronous*), each with its own guarantees on message delivery. For example, causal ordering, where only messages that are related to each other will be delivered in an order that preserves this relationship, and atomic ordering, where messages will always be delivered in the same order to all recipients, but this order may not preserve causality.

The market leaders in the commercial application server space, WebSphere from IBM, WebLogic from BEA and Oracle Application Server, have very similar approaches to clustering. They typically characterize clustering for:

- *Scalability*: The proposed configuration should allow the overall system to service a higher client load than that provided by the simple basic single machine configuration. Ideally, it should be possible to service any given load, simply by adding the appropriate number of machines.
- *Load balancing*: The proposed configurations should ensure that each machine or server in the configuration processes a fair share of the overall client load that is being processed by the system as a whole. Furthermore, if the total load changes over time, the system should adapt itself to maintain this load-balancing property.
- *Failover*: If any one machine or server in the system were to fail for any reason, the system should continue to operate with the remaining servers. The load-balancing property should ensure that the client load gets redistributed to the remaining servers, each of which will henceforth process a proportionately slightly higher percentage of the total load. Transparent failover (failures are masked from a client, who minimally might need to retransmit the current request) is an ideal, but rarely achievable with the current technology, for the reasons to be outlined below. However, the important thing in current systems is that forward progress is possible eventually and in less time than would be the case if only a single machine were used.

Transparent failover is easy to achieve for stateless sessions: any server in the cluster can service any request and if a client makes multiple requests in succession each may well be serviced by a different server. Failover support in this case is trivial: if a failure of the server occurs while it is doing work for the client then the client will get an exceptional response and will have to retransmit the request.

The situation is more complicated for a stateful session. If the same server instance is used for all requests from the client, the server failure will lead to loss of state. The approach adopted in commercial systems to avoid loss of state is to use the stateless session approach with a twist: the session is required to serialize its state to a datastore or one or more application servers within the cluster at the end of each client request and for the subsequent bean instance in the other application server to deserialize the state before servicing the new request. In some cases, the replication of the data is assumed to be the domain of the datastore or cluster itself. In this way, some of the functionality available for stateless sessions can be regained. However, even in this case, a failure during serialization of the session state (which could result in the state being corrupted) is typically not addressed. There is a more serious limitation: transactions cannot be supported: if transactional access to a stateful EJB is used, then the same server instance must be used for every invocation on that bean; in this case, failures may not be recoverable at all.

It's important to realize that there's a difference between replication of the application server and replication of the datastore. The former doesn't require persistence, whereas the latter does. Replicating only the in-memory application server can improve performance over replicating at the level of the datastore, but there's a tradeoff to be made: the ability to tolerate only a subset of types of failure, e.g., power failures and so on.

One system that provided transactional replication capabilities is the Arjuna transaction system that was developed at the University of Newcastle upon Tyne in the mid 1980s. This system was designed to provide a reliable infrastructure for the development of distributed applications and integrated transactions and replication in the manner we described earlier. The Arjuna system has been used for years to run the registration process for the University, tolerating machine failures, network outages and overloads.

So far we've talked mainly about *transaction systems*, or *transaction processing systems*. These enforce ACID properties for transactional resources by using a two-phase commit protocol in conjunction with two-phase locking. But most transaction products are sold as *transaction processing monitors* (TPM). So, just what is a transaction-processing monitor and how does it relate to a transaction system? In the next section we examine the traditional definition of a TPM.

Transaction Processing Monitors

A product that supports the development of transactional applications/systems is often identified as a *Transaction Processing Monitor* (TPM). It typically provides:

- Toolkits and APIs to allow transactions to be demarcated (created and terminated) and controlled in a distributed environment. Given our definition of a transaction system, we can say that it is at the heart of this part of the TPM.

- Integration of security requirements.
- An execution environment to manage transaction load to maintain high throughput as the number of transactions to be executed increases—transactions may be automatically distributed across a number of different execution environments.
- Ensure high-availability for transactions—if a transaction fails it may be automatically restarted on another machine or multiple copies (replicas) may be executed to mask failures.
- Administration services for configuring, monitoring and managing transactions and transactional applications.

In the world of J2EE, many of these services may seem familiar—they form the heart of application servers. A TPM is essentially a transaction-aware application server and in fact many of the features found in J2EE application servers have their basis in transaction processing monitors. Likewise, many modern day TPMs are Java application servers with transaction service cores.

Transactions as we have presented them are sufficient if an application function can be represented as a single top-level transaction. Frequently this is not the case, with application requests requiring more than one transaction. These multi-transactions are typically referred to as *workflows* and are coordinated and controlled by a *workflow system*. Let's now look at workflow systems.

Transactions and Workflow

There are a number of reasons why an application's activities may be structured as multiple independent transactions. Top-level transactions are most suitably viewed as "short-lived" entities, performing stable state changes to the system; they are less well suited for structuring "long-lived" application functions (e.g., running for hours, days or longer). Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources for a long time; further, if such a transaction aborts, much valuable work already performed could be undone.

Workflows are rule-based management software that direct, coordinate and monitor execution of tasks arranged to form *workflow applications* representing business processes. *Tasks* (*activities*) are application specific units of work. A *workflow schema* (*workflow script*) is used to explicitly represent the dependency between the tasks.

The structure of many workflow management facilities is based on the Workflow Reference Model developed by the Workflow Management Coalition (WfMC). Figure 1-29 depicts the Reference Model.

This facility provides for the manipulation and execution of workflow instances (interfaces 2, 3 and 5) as well as for the definition and management of workflow schemas (interfaces 1 and 5).

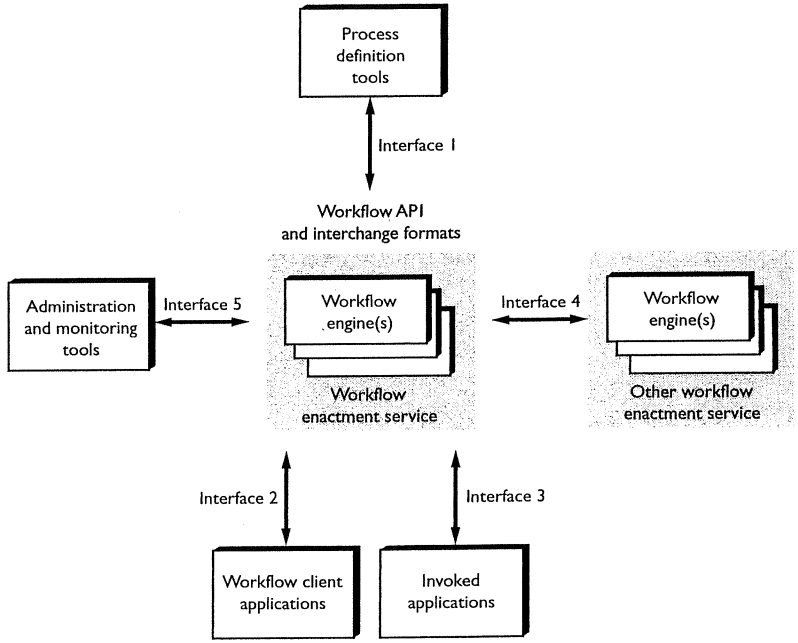

Figure 1-29 WfMC workflow reference model.

According to the base workflow model, a workflow application is modeled as a collection of tasks. A task is the unit of activity within a workflow application; typically you can equate a task with an independent, top-level transaction. The structure of the workflow application is expressed by the interdependencies between its constituent tasks. A dependency could be just a *notification (temporal)* dependency (shown by a dotted line in Figure 1-30, indicating that t2 can start only after t1 has terminated) or a *dataflow* dependency (shown by a solid line, indicating that, say t3, needs to be notified of the availability of input data from t1).

A task is typically modeled as a having a set of *input sets* and a set of *output sets*. The execution of a task is triggered by the availability of an input set. Only the first available input set will trigger the task; the subsequent availability of other input sets will not trigger the task (if multiple input sets became available simultaneously, then the input set with the highest priority may be chosen for processing). For an input set to be available it must have satisfied all of its constituent inputs and notifications.

A task can terminate producing one of a set of output sets, each output set producing a distinct set of outputs. The outputs of the produced output set and the inputs of the first available

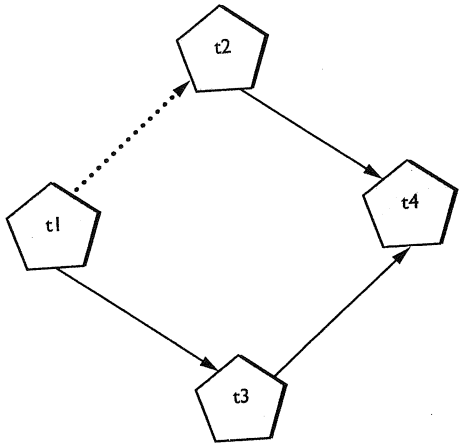

Figure 1-30 Inter-dependency tasks.

input will, if required, be propagated to other tasks as sources of input. In addition, a task can be composed from other tasks to form a compound task.

A workflow system allows the specification of task dependencies and their inter-relationships and then controls the execution of that workflow specification. Typically this specification will be maintained in a persistent manner such that failures of the workflow system or individual tasks will be recoverable to ensure that the required specification is executed to completion.

For example, let's consider a workflow application that involves processing a customer's order for a book from an online shop. It may be modeled as a compound task *processOrderApplication*, which contains four constituent simple task instances: *paymentAuthorisation*, *checkStock*, *dispatch* and *paymentCapture*. The relationship between the tasks is shown in Figure 1-31.

To process an order, *paymentAuthorisation* and *checkStock* tasks (top-level transactions) are executed concurrently. If both complete successfully then *dispatch* task is started and if that task is successful then the *paymentCapture* task is started.

Some workflow implementations allow the internal structure of a compound task to be modified without affecting the tasks that supply it with inputs or use it for inputs. In this case it would be possible to change the payment and stock management policies, for example, causing payment capture even if the item is not presently in stock, or the addition of a task that could check the stock levels of the suppliers of the company and arrange direct dispatch from them.

By now you should have a good grasp of the fundamentals behind transaction processing and many of the subtle (and often not so subtle) issues that arise when distribution, replication, threads and time are added into the equation. As we've seen, something that may initially seem relatively straightforward can quickly become extremely complex to understand, let alone

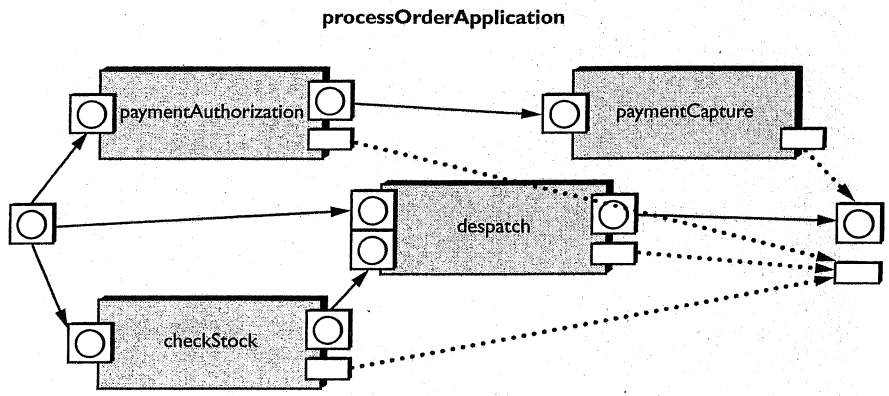

Figure 1-31 Process order application example.

implement. However, transactions have been around for many decades and there have been many commercial implementations and standards. In the following section we briefly look some of the more important events in the transaction timeline.

Summary

In this chapter we examined the main principles behind transactions and saw why they are typically needed. We looked at what ACID properties are and how they may be implemented, including some fairly subtle issues that can affect performance adversely. In order to ensure that all participants see the same outcome, the two-phase commit protocol is necessarily blocking; unfortunately blocking applications and holding on to resources is not practical in many cases so most transaction systems use heuristics to loosen the transaction's hold on resources. Obviously this can have dire consequences on consistency if the heuristic outcome is different than that ultimately taken by the transaction, but heuristics are usually only taken as a last resort.

Failures happen in distributed systems and these failures will often affect transactions. As a result, failure recovery mechanisms are necessary to ensure that transactional semantics occur. However, implementing recovery is extremely complex and most public domain transaction service implementations (and some commercial products) don't provide it.

We also looked at the different types of transactions, including nested and concurrent. Distributed transactions are extremely important and they present some challenging problems for failure recovery and performance. As with failure recovery, interposition is often overlooked by implementations and yet it is a critical component in improving performance.

Today's applications are inherently multi-threaded and it's inevitable that these threads will share transactions at some point. Therefore, we looked at how transactions and threads interact and the issues involved, especially in a distributed environment. Finally, we looked at some advanced transaction issues such as replication to improve availability and workflow systems that allow the gluing together of transactions to form long-running transactional applications.

CHAPTER 2

Foundation of Transactions in J2EE: The Java Transaction API

The transaction concept is powerful because it helps us to guarantee correct outcomes without imposing undo complexity on applications. Historically, different transaction monitors implemented their own variations of transaction processing messages, APIs, and completion routines. And as we've seen, the two-phase commit protocol itself can be implemented with different messages, semantics, and assumptions. Global transactions themselves are useless unless all participants can share the same framework for reaching a consensus. Proprietary implementations often allow unique optimizations for completely homogenous environments, but do not always work well for the complex, heterogeneous and distributed systems that are the hallmark of IT systems in large enterprises today. In order to make the transaction concept useful in the real world, applications, infrastructure and resource managers must share a common basis for coordinating transactional work.

The Java platform provides transactional capabilities by building on the widely adopted Distributed Transaction Processing (DTP) model defined by the Open Group. The leading relational databases, message queues, and even proprietary systems support the specifications based on DTP. Of special interest to Java programmers is the fact that all global transaction processing in the Java 2 Enterprise Edition is defined in terms of this model. In the world of enterprise Java applications, DTP is the cornerstone of transaction processing. So we're going to take a close look at the Open Group transaction model, the specifications that support it, and how they are applied in Java. Most of the technologies that we describe in the rest of the book build on these standards, including the crucial technologies to support relational database access and container managed transaction demarcations. In this chapter we also examine some of the important issues you should be aware of when using a DTP-based transaction manager in the Java environment.