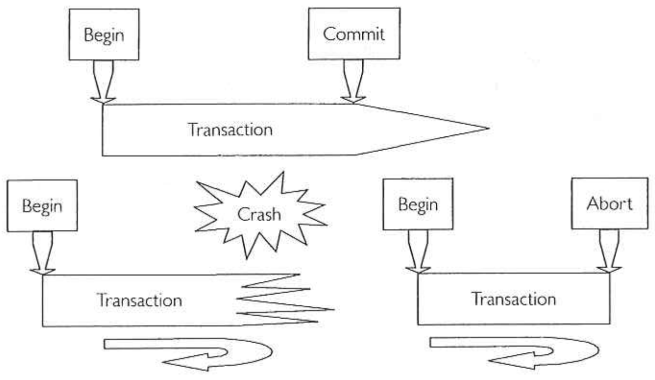
transactions are stored on persistent storage. Together with atomicity, consistency means that transactions can always recover to a consistent point. The combination of consistency and durability means that only consistent information is stored on persistent storage. The combination of isolation and consistency properties implies that no concurrent processes can see the inconsistent information that may arise during the course of a transaction. Transactions can only see the consistent information that was established at previously committed transactions.

11.1.2 Transaction Operations

It is generally client objects that implement a particular application in order to determine the start and end of a transaction. To do so, the client application needs to use a number of operations for transaction manipulation purposes. These operations form the client's perspective on transaction management. Figure 11.1 shows an overview of these operations and their effects.

Figure 11.1
Client Perspective on Transaction Management Operations



Begin starts a transaction.

All operation executions that are requested after a transaction has *begun* are executed under transaction control. Hence, their effect is not visible to concurrent transactions until after the transaction has been finished successfully.

Commit completes a transaction.

During a transaction *commit*, all the changes are stored onto persistent storage and they are made available to other transactions. Moreover, the transaction leaves the resources that it modified in a consistent state.

Abort cancels the effect of a transaction.

A transaction may also terminate unsuccessfully. *Aborts* can either be implicit, due to a failure that terminates one of the participating objects, or explicit by invoking an abort operation. Clients may invoke an abort operation explicitly if they have reached an inconsistent state from which they cannot recover. The effect of implicit and explicit aborts is the same: all changes that the transaction has performed to participating objects are undone so that the state that these objects had before the transaction started is re-established.

Client and server programmers use transaction commands to design transaction boundaries.

We note that the determination of transaction begin, commit and abort is application-specific. Whether client or server objects determine transaction boundaries cannot be defined in general; both are viable options. It is usually the client or server object designer that determines the transaction boundaries and thus client or server object programmers use the transaction commands. We note, however, that the designer of server objects has to inform the client designer whether the operation of the server object defines a transaction or not so as to avoid double use or omission of transaction protection.

11.1.3 Flat versus Nested Transactions

The most basic, and also the most common, form of transaction is a *flat transaction*. This means that a new transaction is only begun if the previous transaction has been either committed or aborted. Flat transactions are easy to implement but they are limited in that an abort leads to the potential loss of all the results that have been achieved during the course of the transaction.

Alternating begin and commit commands lead to flat transactions.

Nested transactions overcome this limitation. A *nested transaction* is embedded in another transaction. This other transaction is referred to as *parent transaction* while the nested transaction is sometimes also called a *child transaction*. Nested transactions are used to set continuation points during the course of the parent transaction so that parts of a more complex transaction can be undone without losing all changes. This is achieved by the following execution semantics of the transaction.

A nested transaction is embedded in another transaction.

Child transactions can commit or abort independently of their parent transactions. Figure 11.2 visualizes this behaviour. If a child transaction aborts, only those changes that were done within the child transaction are undone. If a child transaction is completed, the changes are made visible to the parent transaction and successive sibling transactions. However, if a parent transaction aborts, all the changes made by its child transactions will be undone.

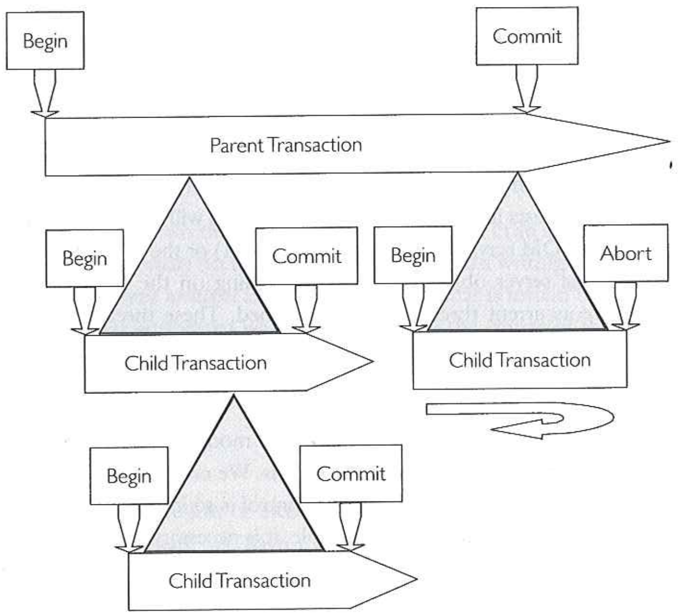


Figure 11.2
Client Perspective on Nested Transactions

Nested transactions provide flexibility to designers but are difficult to implement.

Hence, nested transaction give additional degrees of freedom to the designer of an application. It should, however, be noted that the additional flexibility gained by nested transactions does not come for free. It is considerably more complicated to implement the concurrency control and commit protocol of nested transaction than those needed for flat transactions.

11.2 Concurrency Control

After having introduced the transaction concepts, we will now switch perspective and review how transactions can be implemented. We will start with concurrency control techniques, which are used to achieve the isolation property of transactions.

Concurrency control is transparent to client designers, but may not be transparent to server designers.

The application of concurrency control techniques is transparent to designers of transactional clients. Whether or not it is also transparent to the designers of transactional servers depends on the way persistence is achieved. If relational or object databases are used to achieve persistence, then the concurrency control mechanisms of these database systems are generally sufficient to guarantee isolation. If persistence is achieved in a different way, for example by storing state information in files then the server object designer has to implement concurrency control manually and the techniques discussed here should be applied.

Before we study the concurrency control techniques in detail, we will first investigate why concurrency control is necessary. We will review two typical issues that make concurrency control necessary, lost updates and inconsistent analysis. We will then study two-phase locking, which is the most commonly-used concurrency control technique in databases as well as for distributed objects. We will then see that deadlocks can occur in two-phase locking and we will investigate how to detect and resolve deadlocks. We will finally discuss hierarchical locking techniques and see how concurrency control is supported by the CORBA concurrency control service.

11.2.1 Motivation

Servers usually have more than one client object. It can, therefore, happen that these client objects make object requests in parallel. The object requests will then arrive at the CORBA object adapter or the COM service control manager (SCM) or the Java activation interfaces on the host where the server object executes. Depending on the configuration of these adapters or SCMs, concurrent threads may be spawned. These threads are then used to execute the requested operations of the server object concurrently. In order to make these points more clear, we use an example.

If the requested server objects include operations that modify the server's state, two problems may occur: lost updates and inconsistent analysis. We now investigate these problems in detail to provide a reason for why concurrency control is so important. In order to study the concurrency control problems with our example, it is necessary to make assumptions on the way that the methods of `Account` are actually implemented. Figure 11.8 shows an example implementation.

```
class Account {
 private:
  float balance;
 public:
  float get_balance() {return balance;};
  void debit(float amount){
    float new=balance-amount;
    balance=new;
  };
  void credit(float amount) {
   float new=balance+amount;
   balance=new;
  };
};
```

This class declaration refines the `Account` class designed in the class diagram of Example 11.2.

Example 11.8
Implementation of `Account` Operations

Lost Updates

A *schedule* is a particular interleaving of operations of concurrent threads. Generally concurrent execution allows many schedules to occur and the aim of concurrency control is to reject those schedules that lead to unwanted interference between the concurrent processor threads. Lost updates are an interference that we wish to avoid.

A schedule results in a *lost update* if one concurrent thread overwrites changes that were performed by a different thread. Lost updates are not confined to distributed objects or databases, but they can occur in concurrent programs in general. Example 11.9 shows an occurrence of such a lost update.

Lost updates are modifications done in one thread that are overwritten by another.

Inconsistent Analysis

Lost updates occur when two threads modify a shared object in an improper way. A different form of concurrency anomaly occurs between a reading and a writing thread. The anomaly is referred to as *inconsistent analysis* and denotes a result that is invalid because the object on which the analysis was performed was modified during the course of the analysis. Example 11.10 shows a schedule leading to an inconsistent analysis.

Inconsistent analysis occurs when an object is modified concurrently while the analysis is performed.

Serializability

A system where updates are lost or incorrect results are produced is clearly undesirable. In order to solve this problem, we have to control concurrency in such a way that those schedules that cause lost updates or inconsistent analysis are avoided. In order to characterize such schedules we define the concept of serializability.