11

Distributed Object Transactions

Learning Objectives

In this chapter, we study the concepts of concurrency control and transactions. We will comprehend why we need to perform concurrency control in a distributed object system. We will acquaint ourselves with two-phase locking, which is the basic concurrency control protocol. We will study the concurrency control primitives that are available in object-oriented middleware. We will then familiarize ourselves with the properties of a transaction and investigate how distributed object transactions can be implemented. We will then investigate the relationship between concurrency control and distributed object transactions and realize that concurrency control is needed in order to implement the isolation property of transactions. We will then study the two-phase commit protocol and understand how it implements the atomicity property of transactions. We investigate the distributed transaction services provided by object-oriented middleware and comprehend how designers use these services.

Chapter Outline

We have discussed in the introductory chapters that distributed systems may have multiple points of failures. While centralized systems either do or do not work, distributed object systems may work only partially, which may lead to integrity violations.

Example 11.1
Consequences of Failures

A good example is a funds transfer from an account in one bank to an account in another bank. The banks use different hosts for their account management objects. In an Internet banking application, the object from which the funds transfer is initiated resides on yet another host, for example the client's home PC. The funds transfer involves executing a debit operation on the account object from which the funds are transferred and a credit operation on the account object of the target bank. If one account-holding object is unavailable to serve requests, only one of the operations will succeed and the other one will fail. This will result in either funds being lost or funds being generated. Neither situation is acceptable.

Integrity violations can also occur due to concurrent accesses to objects. If two clients invoke operations concurrently and the object adapter used on the server-side supports concurrent execution of requests, we may find situations where changes performed by one object are overwritten by another object. To prevent such integrity violations we have to control the concurrent execution and only permit those schedules that respect the integrity of objects' data.

Transactions are the main concept that we introduce in this chapter. Transactions cluster several object requests into a coarse-grained unit for which a number of properties hold: the transaction is either performed completely or not at all; the transaction leaves objects in a consistent state; the transaction is performed in isolation from any concurrent transaction; and once completed, the result of the transaction is prevented from being lost.

In the last chapter, we saw how objects can make their states persistent. The ability to retain state on persistent storage is a powerful measure in the struggle against failures. We will see that transactions utilize the ability to persistently store state information. We will first introduce the main principles and concepts of transactions. In the second section we discuss the techniques for concurrency control that a transaction uses in order to guarantee an execution of its requests in isolation from any concurrent transactions. We then discuss that at least two phases are needed to implement the commit of a transaction in a distributed setting. We conclude by reviewing the technologies that object-oriented middleware provides for implementing distributed object transactions.

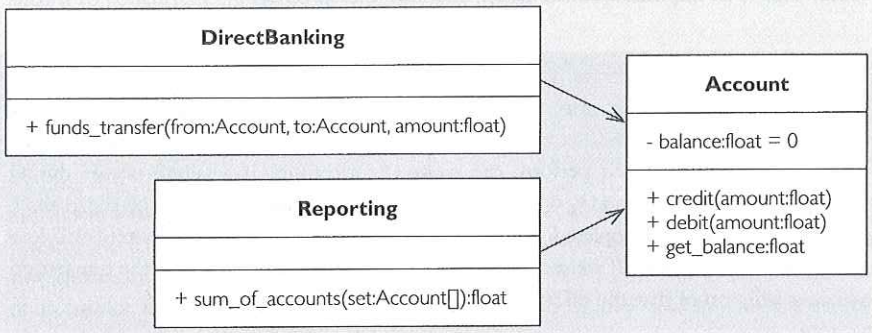
11.1 Transaction Principles

11.1.1 Concepts

Before we can see how transactions can be implemented, we need to understand what exactly transactions are. We also need to develop a feeling for when they are used. We therefore first define a transaction and its defining properties. We then discuss flat versus nested transactions and we classify the roles that objects can play in transactions.

A *transaction* clusters a sequence of object requests together in such a way that they are performed with ACID properties. This means that the transaction is either performed completely or not at all; it leads from one consistent state to another one; it is executed in isolation from other transactions; and, once completed, it is durable.

A transaction is an atomic, consistent, isolated and durable sequence of operations.



Example 11.2
Types of Objects in Banking Example

This class diagram shows the types of objects that participate in example transactions that we use for illustration purposes throughout this chapter. The examples will use objects of type `Account`, which have an internal state. The state captures the balance of the account as a float number. Objects of type `Account` can `debit` or `credit` a sum to the account and use `get_balance` to find out what the account currently holds.

Example 11.3
Funds Transfer as Transaction

A funds transfer, which involves requesting a `debit` operation from one account and a `credit` operation from another account, would be regarded as a transaction. Both of these operations have to be executed together or not at all; they leave the system in a consistent state; they should be isolated from other transactions; and they should be durable, once the transaction is completed.

As Example 11.3 suggests, more than one object may be involved in a transaction and the object requests modify the states of these objects. However, a sequence of read operations could also be executed as a transaction. These operations do not perform any modifications, but it may be necessary to execute them as a transaction to ensure that they are not modified while the reading goes on. Hence, transactions allow us to attach particular execution semantics to a sequence of object requests. This semantics is characterized by the ACID properties that we discuss now.

The execution semantics of transactions cannot be defined for single operations.

Atomicity

Atomicity denotes the property of a transaction to be executed either completely or not at all. When a failure occurs in the middle of a sequence of object requests, some object request may have been completed while others will never be completed due to the failure. If the sequence of requests is executed as a transaction, the transaction manager will ensure that the effect of those requests that have been completed are undone and that the state of the

A transaction is either executed completely or not at all.

distributed object system is restored to the situation in which it was before the transaction was started.

Atomicity is an important concept because it enables application programmers to use transactions to define points to which the system recovers in a transparent way if failures occur. This is an important contribution to failure transparency.

Example 11.4
Atomicity of Funds Transfer Transaction

To implement a funds transfer, we need to perform a `debit` operation on one account object and a `credit` operation on another account object. Usually these operations are executed in sequential order. Let us assume without loss of generality that we start with the `debit` operation and then perform the `credit` operation. If a failure occurs during the execution of the `credit` operation, we must also undo the effect of the `debit` operation, which was completed successfully. If we do not do this the system will lose money due to the failure. If we execute the two operations as a transaction, the transaction manager will ensure that the effect of the `debit` operation is undone.

Consistency

A transaction preserves consistency.

In every system there are application-specific consistency constraints that must not be violated. Transactions preserve *consistency* in that they lead from one consistent state to another one. This does not mean that inconsistencies do not occur, but they are confined within the boundaries of one transaction. If a transaction does not reach a consistent state it cannot be allowed to commit; it has to be aborted and all previous changes have to be undone.

Example 11.5
Consistency in Funds Transfer Transaction

In our funds transfer transaction, a consistency constraint would be that banks do not lose or generate monies during their operation. Hence, every account management transaction has to ensure that it credits monies to accounts in exactly the same amount as it debits monies. Note also that inconsistencies are bound to occur during the transaction. There is a point in time when an amount has been debited but the credit operation is still outstanding. Hence, the transaction temporarily violates the consistency constraint.

We have seen in the preceding chapters that consistency can also be defined at an object level. An account object can, for example, reject execution of a debit operation if it would leave a negative balance (or a balance that exceeds some previously-agreed overdraft limit). One object, however, can usually not decide on the consistency of more global information that spans multiple objects. In our example, this would be that the sum of balances on all account objects is equal to the assets of the bank. It is for these more global consistency constraints that transactions are needed.

Consistency is such an important concept of a transaction because it gives designers the assurance that, once a transaction is completed, it has left the system in a consistent state. Transactions provide this mechanism to define consistency constraints among all the objects that are accessed during a transaction.

Isolation

Transactions are performed in isolation from any other concurrent transactions. The *isolation* property of a transaction means that there cannot be any interference between two concurrent transactions. This means that no changes that a transaction does to a system are shown to concurrent transactions until the transaction is complete.

Transactions a
isolated from e
other.

The transaction manager ensures that two transactions that attempt to access the same accounts concurrently are forced to a sequential execution schedule so as to exclude integrity violations.

Example
Isolation of Fu
Transactions

To achieve isolation property, transaction managers have to perform concurrency control and this motivates the next section, where we will discuss the concurrency control problem and its solutions in more detail. Note, however, that isolation does not mean that transactions are not executed concurrently at all. This would be highly undesirable because distributed objects are potentially used by a high number of concurrent users. In the bank system example that we presented in Chapter 1, there are about 2,000 employees that manipulate account information concurrently. It would be unsatisfactory if their accesses had to be performed in a strictly sequential order. The focus of the isolation property is on avoiding interference.

The isolation property of transactions provides concurrency transparency (see Page 52). In particular, it means that application programmers that use transactions are guaranteed exclusive access to all participating objects without taking any further measures.

Durability

Once a transaction has successfully been completed, its changes persist and cannot be undone. This *durability* is usually achieved by integrating distributed objects with the persistent state services that we discussed in Chapter 10. The difference is that now it is the transaction manager rather than the object adapter or service control manager that calls upon the service to make persistent the states of all objects that participated in a transaction.

The changes c
completed tra
persist.

Upon completion of the funds transfer transaction, the account objects would need to store their changed balance attributes onto persistent storage. To do so, they would utilize a storage object that could either be stored in a file or in a relational or an object database.

Example
Durability in F
Transaction

Durability is an important property of a transaction because it means that the processes that execute objects may deliberately or accidentally terminate after the transaction and the changes that the transaction made are still preserved.

Summary

We note that the power of transactions lies in the combination of these ACID properties. The combination of atomicity and durability means that only the effect of completed