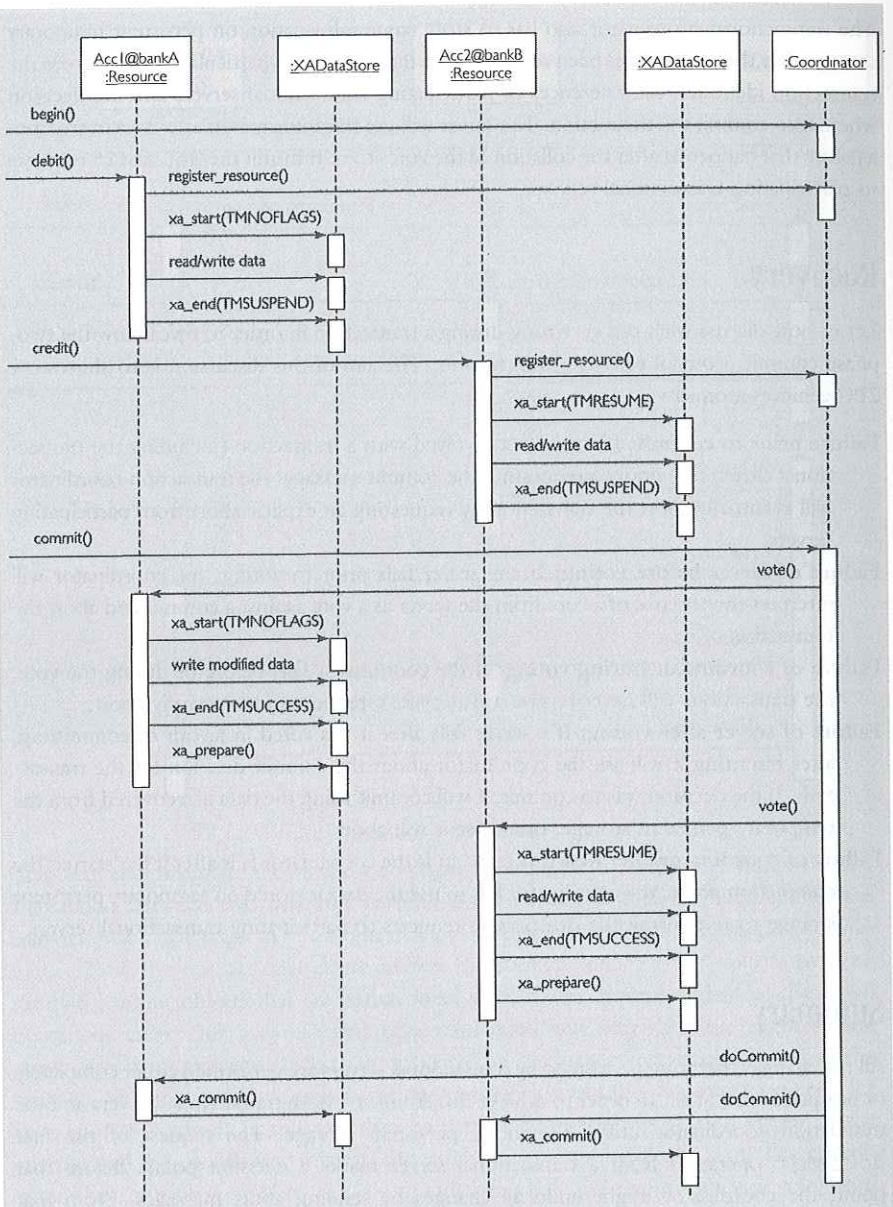
Example 11.16

Two-Phase Commit Implementation using ODTP/XA Protocol



The above diagram shows the interactions between transactional servers and different databases that are used to store persistently the state of the account objects of Example 11.15. Access or update operations of the state of account objects are encapsulated by `xa_start` and `xa_end` operations so that the database uses its own locking mechanisms to achieve the isolation property. Note how the implementation of the voting phase uses the database to store the modified data temporarily. The database then uses its own mechanisms to make those changes permanent and then release all locks on the accessed and modified data during the `xa_commit` operation.

can we use the two-phase commit protocol to implement distributed object transactions between these databases?

Fortunately all database vendors have understood the importance of distributed transaction processing. In addition to their internal transaction processing capabilities, they offer interfaces so that their transaction execution can be externally controlled. This enables databases to participate in transaction implementations that follow the two-phase commit protocol. In order to facilitate the portability and interoperability, a standard interface has been defined for Open Distributed Transaction Processing (ODTP) by the Open Group.

The XA protocol is part of the ODTP standard and defines the application programming interface that database management systems have to implement for transaction processing. In particular, it includes operations to start a transaction (`xa_start`), to end or suspend a transaction (`xa_end`), to vote on committing a transaction (`xa_prepare`) and to do the commit (`xa_commit`). The XA protocol standardizes the parameterization and the semantics of these operations. Example 11.16 shows how this XA protocol is used in a transactional server implementation.

Thus, transactional servers that delegate persistence to a DBMS will use the XA protocol for also delegating the implementation of the two-phase commit operations. This delegation will considerably simplify the implementation of transactions in transactional servers and should be chosen by server object designers whenever possible.

Servers can delegate transaction implementations to a DBMS.

11.4 Services for Distributed Object Transactions

We have explained the principles of distributed object transactions. We have identified how the isolation property can be implemented in those cases when it is not addressed already by a database management system. We have shown how the atomicity property of transactions is implemented using the two-phase commit protocol. We will now review standard technologies that are available for the implementation of distributed object transactions.

In this section, we discuss the CORBA Transaction Service that is used for implementing CORBA-based distributed object transactions. We then see how COM objects can implement transactions using the Microsoft Transaction Service and finally we review Sun's Java Transaction Service that is used by distributed Java objects.

11.4.1 CORBA Transaction Service

The CORBA Transaction Service was adopted as part of the CORBAservices RFP2 in 1994. It standardizes CORBA interfaces to implement the principles that we have discussed in this chapter. We now discuss excerpts of these interfaces in order to show how a sequence of CORBA object requests can be executed with transaction semantics.

CORBA Transaction Architecture

Figure 11.8 shows an overview of the interfaces of the CORBA Transaction service. For reasons of brevity, the figure includes only those operations that show that the Transaction Service implements the principles that we discussed above. We note also that we have omitted a number of less important interfaces.

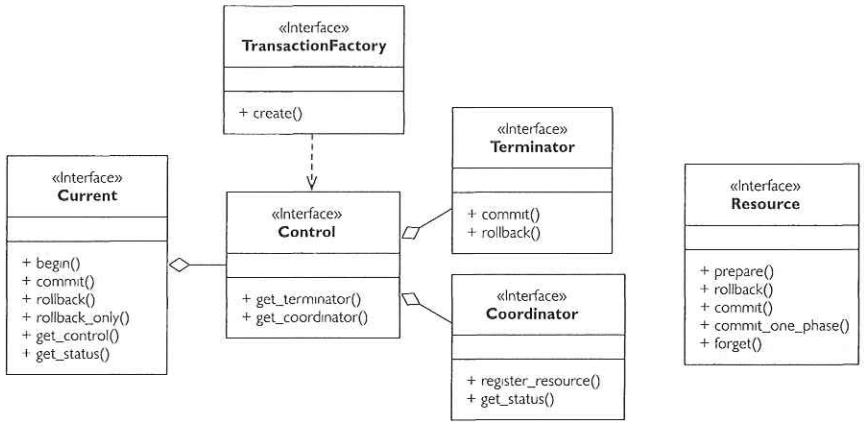


Figure 11.8
Interfaces of the CORBA Transaction Service

The CORBA transaction service uses the two-phase commit protocol for transaction management. The operations for transaction coordination purposes are distributed among several CORBA interfaces in order to provide dedicated interfaces for transactional clients and transactional servers. Every CORBA thread has an implicit current transaction associated with it. If implicit transaction contexts are used, the transaction operations are provided by the `Current` object. In addition to the implicit transaction control, CORBA supports the creation of explicit transaction contexts. They are created using the Transaction service's `TransactionFactory`, a reference to which can be obtained using the `FactoryFinder` interface of the cycle service. Transactional clients can start a new transaction using the `create` operation and are given a reference onto a transaction control object, which can pass a reference on a transaction terminator object. The terminator object is then used to commit or rollback (that is, abort) a transaction.

The `Coordinator` and the `Resource` interfaces are relevant for transactional servers. CORBA objects that are transactional servers register their involvement in a transaction using the `register_resource` operation to which they pass a reference to themselves as a parameter. CORBA transactional server objects have to implement the `Resource` interface. It includes the operations needed for the two-phase commit. Their vote is requested using the `prepare` operation and requesting `commit` demands the server to commit the transaction. The `rollback` operation is requested from a server in order to abort a transaction and `forget` is invoked when the server can forget all knowledge about the transaction.

Using CORBA Transactions

There are two ways that transactional clients can start a transaction. Transactional clients can obtain a reference on an object representing the current transaction using the CORBA

initialization operation `resolve_initial_references`, which returns an object of type `Current`. They then use the operations provided by `Current` to start, commit or abort a transaction. The second way to start a transaction is to create a new `Control` object using the `TransactionFactory`. The transaction is then completed using a `Terminator` object to which the client can obtain an object reference from the `Control` object.

To implement transactional servers in CORBA, designers need to implement the `Resource` interface. In particular, they need to implement the `prepare`, `commit` and `rollback` operations. If a relational or object database is used for persistence, they can delegate these calls using the XA protocol to the DBMS.

11.4.2 Microsoft Transaction Server

A sequence of COM operation executions can be executed as a transaction using the Microsoft Transaction Server (MTS). Like the CORBA Transaction service, MTS uses the two-phase commit protocol for the execution of distributed transactions. MTS originates in the SQL Server relational database product of Microsoft, but it will become part of the next generation of the Windows operating system.

MTS Architecture

Figure 11.9 provides an overview of the main components of MTS. The transaction coordination is performed in MTS by the *Distributed Transaction Coordinator (DTC)*. There is one DTC object on every MTS. The DTC manages the transaction context objects on behalf of MTS.

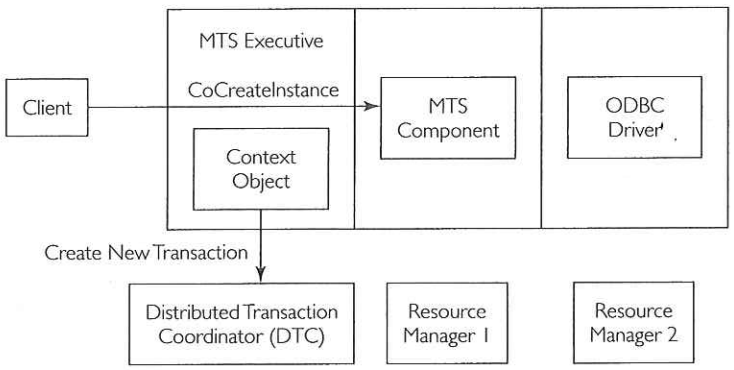


Figure 11.9
Distributed Object Transactions using MTS

There is one *context object* for every transaction. The context object implements the interface that is used by MTS Components in order to identify whether they are able to commit or whether a transaction should be aborted. The context object also provides a redefined `CreateInstance` operation that facilitates the creation of COM components that execute under MTS control.