MTS Components are the equivalent to resources in the CORBA Transaction service. Unlike CORBA resources, however, they do not have to implement a particular interface. They use the context object to indicate whether they can commit transactions or want to abort the transaction in which they participate.

Resource Managers

Resource managers store the state of MTS components persistently and therefore they participate in the two-phase commit protocol. When an MTS Component calls a resource manager for the first time, the resource manager registers itself with the MTS. This registration is referred to as *enlistment*, in Microsoft's terminology. From then on, the MTS will ensure that the resource manager is involved in the two-phase commit protocol. At the time of writing this book, the only type of resource manager that was supported by MTS was Microsoft's SQL Server product. It can, however, be expected that MTS will soon support other resource managers, in particular database management systems implementing the XA protocol and the NT file system.

Using MTS Transactions

The use of MTS is more transparent to client and server programmers than the use of the CORBA Transaction service. COM clients that use MTS components are generally unaware of transactions. The transactions are started by the MTS based on transaction attributes that are established using MTS Explorer, the administration tool that is provided for MTS.

In order to write transactional servers, COM designers need to build an MTS component. MTS components have to be in-process servers (also known as DLLs). The component needs to have a class factory that MTS can use for creating instances of the component. Moreover, the component must interface with the context object in order to indicate whether it is ready to commit a transaction or whether the transaction should be aborted.

11.4.3 Transactions in Java

Sun recently released the Enterprise Java Beans (EJB) specification. Enterprise Java Beans are Java Beans that use RMI to communicate with clients and other beans across distributed Java VMs. Because client applications may wish to execute a series of different EJB operations in an atomic, consistency-preserving, isolated and durable fashion, the EJB specification demands provision of transactions. In order to support the implementation of such transactions, Sun defined the Java Transaction API and the Java Transaction Service. The Java Transaction API defines the interfaces that are used by transactional clients and transactional servers. EJB providers use this Java interface to implement transactions. The Java Transaction Service defines how transaction coordinators can be implemented based on the CORBA Transaction service.

Java Transaction Architecture

Figure 11.10 shows an overview of the different layers of the Transaction Management architecture for Java.

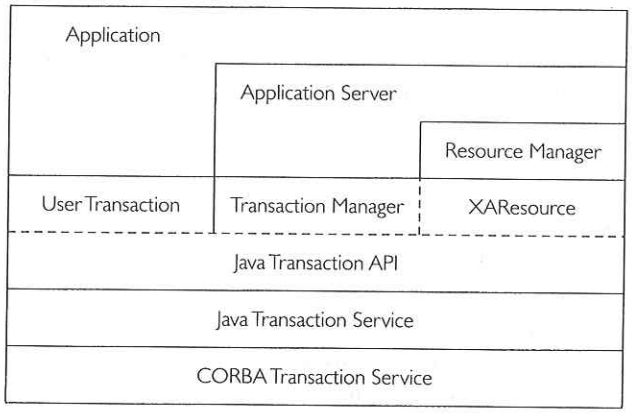

Figure 11.10
Distributed Transaction Management in Java

The Java transaction management uses the two-phase commit protocol for transaction management purposes and defines three main interfaces in the Java Transaction API (JTA). The `UserTransaction` interface is used by applications in order to control transactions. The `TransactionManager` interface is implemented by application servers, such as transaction monitors or Enterprise Java Beans implementations in order to provide a transaction coordinator. Finally, the `XAResource` interface is implemented by transactional servers. Figure 11.11 shows an excerpt of the operations provided by the most important interfaces of the JTA.

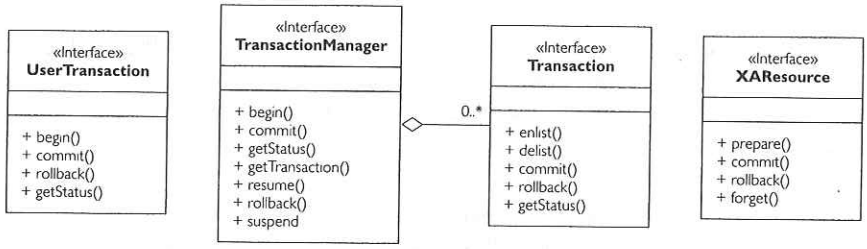

Figure 11.11
Interfaces of the Java Transaction API

`UserTransaction` provides the operation for a transactional client to start new transactions (`begin`), to commit a transaction (`commit`) and to abort a transaction (`rollback`). In fact, the purpose of `UserTransaction` in the Java Transaction API is identical to interface `Current` in the CORBA Transaction service. `TransactionManager` is the coordinator interface. It provides operations to implement the transaction begin and end. Implementations of `UserTransaction` will delegate transaction management operations to `TransactionManager`. The `TransactionManager` interface also provides a mechanism to return a reference to one of the `Transaction` objects that it manages. The `Transaction` interface has the same purpose as the `Coordinator` interface of the CORBA Transaction service. The `enlist` operation of a `Transaction` object is used by a transactional server to register its participation in a transaction. Finally, the `XAResource` interface corresponds to the `Resource` interface of the CORBA Transaction service and has to be implemented by all transactional servers that wish to participate in two-phase commit transactions.

Using Java Transactions

The use of the Java Transaction API is very similar to the use of the CORBA Transaction service. They both support the XA protocol and the Java Transaction API can be seen as an extension of the CORBA Transaction service.

The transactional client obtains a reference onto the `UserTransaction` object using an interface provided by the JTA. It can then use `UserTransaction` operations to control start, commit and abort transactions.

A transactional server has to implement the `XAResource` interface and it will most often do so by delegating calls to an XA-capable database. Moreover, it has to register its involvement with the transaction coordinator using the `enlist` operation that is available through the `Transaction` interface. The resource can obtain a reference to its `Transaction` object using the `TransactionManager`.

Summary

Distributed object transactions are well supported by object-oriented middleware.

We have seen that all object-oriented middleware systems support transaction management through dedicated services. All of these services offer an integration with database systems. In the case of CORBA and Java this integration is the XA industry standard, which is not yet supported by MTS. MTS uses a proprietary protocol for an integration with SQL Server.

Transaction services are integrated with DBMSs using the XA protocol.

The isolation property is generally achieved using the locking mechanisms supported by the database. In cases where no database is used, the designer of a transactional server has to use concurrency control mechanisms, such as the CORBA Concurrency service to avoid lost updates and inconsistent analysis.

Middleware uses two-phase commit to implement transactions.

We have also seen that all transaction services use the two-phase commit protocol that we have explained in this chapter. They all standardize interfaces for transactional servers to register their involvement in a transaction and then the servers are called upon by the transaction coordinator to vote and to do the commit.

Key Points

- In this chapter we have introduced the basic principles of distributed transaction management. A transaction is an atomic, consistency-preserving, isolated and durable sequence of object requests.
- The objects involved in distributed transaction management play different roles. A transactional client determines the beginning and end of the transaction. A transactional server maintains a state and performs modifications to that state under transaction control. A transaction coordinator determines whether or not a transaction can be committed using the two-phase commit protocol.

- The implementation of transactions involves concurrency control, which can be done explicitly by a transactional server designer or it can be delegated to a database system.
- Explicit concurrency control is implemented using the two-phase locking protocol. It achieves serializability, but is not deadlock free.
- Deadlocks are detected by a concurrency control manager and they are resolved by aborting a transaction involved in the deadlock.
- The two-phase commit protocol consists of a voting phase and an implementation phase. During voting, the transaction coordinator checks whether every participating transactional server is able to commit the transaction. During commit, the servers save their changes to persistent storage and thus implement durability.
- The phase after a transactional server has voted for a commit and before it receives the commit request is called server uncertainty. It has to be covered by storing the modifications of the transaction on temporary persistent storage.
- Object-oriented middleware provides services for the implementation of transactions. We have discussed the CORBA Transaction service, MTS and the Java Transaction API. All of these services use the two-phase commit protocol and an integration with database systems for the implementation of transactions.

Self Assessment

11.1 What are the ACID properties of transactions?
11.2 What is the difference between two-phase locking and two-phase commit?
11.3 What is lock compatibility?
11.4 When is the hierarchical locking protocol applied?
11.5 What is the purpose of upgrade locks?
11.6 Why are upgrade locks mutually incompatible?
11.7 For which designers is concurrency control transparent?
11.8 When do designers of transactional servers not have to implement concurrency control?
11.9 How does 2PL prevent inconsistent analysis?
11.10 What are the three roles that objects can play in distributed object transactions?
11.11 Why do distributed transactions need two phases to commit?
11.12 How are the ACID properties implemented in distributed object transactions?
11.13 What is server uncertainty?
11.14 How is server uncertainty dealt with?
11.15 Of which tasks do the CORBA Transaction Service, MTS and JTS relieve the designer?

Further Reading

The idea of transactions and the four ACID properties were first introduced by [Gray, 1978] in a course on operating systems. [Gray, 1981] reports on the application of transactions in databases, an area where they have been most successful.