C H A P T E R 7

Transaction Management

by Tony Ng

TRANSACTIONS are a mechanism for simplifying the development of distributed multiuser enterprise applications. They are also one of the standard services offered by the J2EE platform. By enforcing strict rules on an application's ability to access and update data, transactions ensure data integrity. A transactional system ensures that a unit of work either fully completes or has no effect at all. Transactions free an application programmer from dealing with the complex issues of failure recovery and multiuser programming.

The chapter begins with a general overview of transactional concepts and J2EE platform support for transactions. Then it describes the Java Transaction Architecture, the interface used by the J2EE platform to manage and coordinate transactions. Finally, the chapter describes the transactional models available to the each type of J2EE component and to enterprise information systems.

7.1 Transactional Concepts

A *transaction* is a logical unit of work that either modifies some state, performs a set of operations, or both. An individual transaction may involve multiple data and logical operations, but these operations always occur as an indivisible atomic unit, or they do not occur at all. For example, a grocery store cashier totals the prices for a customer's goods, adds the appropriate taxes and subtracts discounts, collects coupons and payment, dispenses change, and gives the customer a receipt. All of the activities described are possibly optional subtasks of a single transaction, failure of any one of which (customer has no money, cash register broken, etc.) causes the entire transaction to fail.

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

A transaction that involves multiple processing nodes, threads of control, and/or information resources is called a *distributed transaction*. Distributed transactions are atomic groupings of data transformations and logical operations, and may also imply that these operations occur in a particular order. For example, enrolling a patient in a health care plan may involve first acquiring release forms from a patient, and only then verifying the patient's employment, checking her health and insurance history against remote data sources, and so on.

Distributed transactions also imply concurrency, which introduces the possibility that the same state represented in two separate places may conflict. The special synchronization, latency, and data integrity properties of distributed transactions make them more complex than non-distributed transactions.

This section provides a brief introduction to basic concepts in conventional and distributed transactional systems. See the Resources section at the end of this chapter for references to in-depth treatment of these topics.

7.1.1 ACID Transaction Properties

Enterprise transactions share the properties of *atomicity*, *consistency*, *isolation*, and *durability*, denoted by the acronym ACID. These properties are necessary to ensure safe data sharing.

Atomicity means that a transaction is considered complete if and only if all of its operations were performed successfully. If any operation in a transaction fails, the transaction fails. In the health care example described above, a patient can be enrolled only if all required procedures complete successfully, so enrollment is atomic.

Consistency means that a transaction must transition data from one consistent state to another, preserving the data's semantic and referential integrity. For example, if every health care policy in a database requires both a patient to be covered and a plan describing the coverage, every transaction in the health insurance application must enforce this consistency rule. While applications should always preserve data consistency, many databases provide ways to specify integrity and value constraints so that attempted transactions that would violate consistency automatically fail.

Isolation means that any changes made to data by a transaction are invisible to other concurrent transactions until the transaction commits. Isolation requires that several concurrent transactions must produce the same results in the data as those same transactions executed serially, in some (unspecified) order.In the health plan

E A R L Y D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

enrollment example, two people updating the same patient's information will not see each others' changes until either commits their changes to the database.

Durability means that committed updates are permanent. Failures that occur after a commit cause no loss of data. Durability also implies that data for all committed transactions can be recovered after a system or media failure.

An ACID transaction ensures that persistent data always conform to their schema, that a series of operations can assume a stable set of inputs and working data, and that persistent data changes be recoverable after system failure.

7.1.2 Transaction Participants

An application that uses transactions is called a *transactional application*. In the J2EE architecture, a transactional application may be either J2EE application server, such as an EJB server, or some type of standalone client, such as an application client. As shown in Figure 7.1, an application accesses external resources by way of a *resource manager*. A resource manager provides and enforces the ACID transaction properties for a specific resource. The resource manager implements transaction demarcation requests (see below) and transactional RPC in terms of a protocol specific to the resource it manages. Examples of resource managers include a relational database (which manages persistent storage), an EIS system (managing transactional, external functionality and data), and the Java Messaging Service (which manages transactional message delivery).

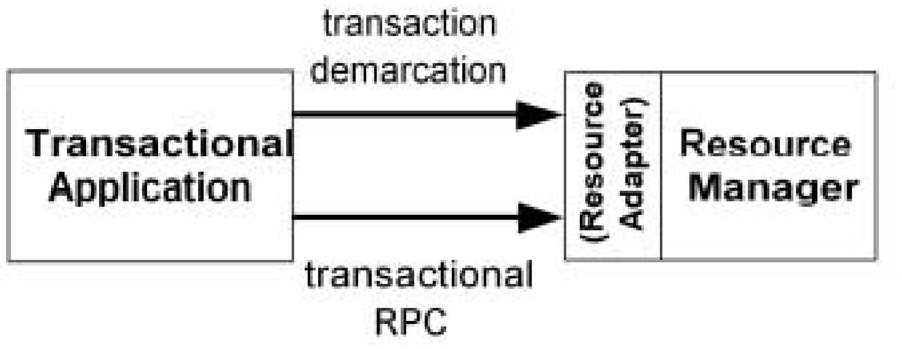

Figure 7.1 Resource managers provide applications with access to external resources

An application accesses a resource manager by way of a system library called a *resource adapter*. Resource adapters tend to be specific to a particular resource manager or resource manager type. For example, a JDBC driver is a generalized resource adapter for a database. An application indirectly accesses a database (the

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

“Resource Manager” in Figure 7.1) by way of the JDBC driver (the “Resource Adapter”), instead of directly accessing the database through a low-level API. In this chapter, resource adapters are treated as part of the resource manager.

A Java Connector Extension is another example of a resource adapter. The Java Connector Architecture, described in the EIS tier chapter of this book, is a general API to which EIS system vendors can write standardized resource adapters.

7.1.3 Transaction Demarcation

Transactional programs must be able to start and end transactions, and be able to indicate whether data changes are to be made permanent or discarded. Indicating transaction boundaries for a program is called *transaction demarcation.* A program using transactions starts a new transaction with a *begin* operation, and ends the transaction with either *commit* (to persist data changes) or *rollback* (to discard changes). A state diagram for a resource manager’s behavior in response to these demarcation operations appears in Figure 7.2.

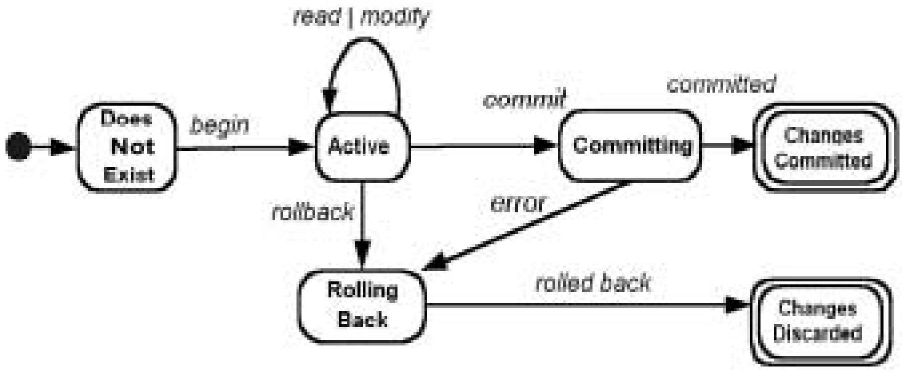

Figure 7.2 State transition diagram for a non-distributed transaction

In the figure above, a program executes a *begin* operation to start a transaction. The program continues reading and/or modifying data under the scope of the active transaction. When the program is ready to make its data changes permanent, it executes a *commit* operation, causing the transaction to persist any data modified or created during the active state. Successful completion of the commit operation results in a permanent change to the transactional resource. If the opera-

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

tion of committing the data changes fails for some reason (inadequate resources, data consistency violations, etc.), the resource manager executes a *rollback*, discarding any changes made since the transaction began. An application may also explictly request a rollback during an active transaction.

J2EE technology provides two ways of specifying transaction demarcation: programmatically (meaning explicitly, in code) or declaratively (implicity by way of a deployment descriptor).

7.1.4 Transactional Semantics

The transactional behaviors defined for a particular resource manager type are called the resource manager's *transactional semantics*. For example, enterprise beans use defined attributes in deployment descriptors to declaratively demarcate a component's transaction boundaries. The transactional behaviors defined for each attribute are that attribute's transactional semantics.

Each type of resource adapter has different transactional semantics because of the properties of the resource. The term "rollback" has a somewhat different meaning for database connections, EIS resource adapters, and JMS providers. Each of the sections later in this chapter on these resource adapter types include a discussion of that adapter type's transactional semantics.

7.1.5 Distributed Transactions

Distributed enterprise systems often need to access and update multiple separate transactional resources in order to accomplish some business goal. Consider, for example, a travel agency application. Creation of a typical business travel intinerary with a confirmed and paid plane ticket requires successful completion of user authentication, credit card processing, and flight reservation, as well as local creation of the itinerary itself. Such a transaction, involving independent, cooperating transactional systems, is called a *distributed transaction*.

Distributed transactions are more complex than non-distributed transactions because of latency, potential failure of one or more resource managers, and interoperability concerns. On a network, a failed transaction be difficult to distinguish from one that is merely slow. Resource managers (as shown in Figure 7.1) that don't "know" about each other cannot coordinate transactions by themselves. While a transactional application could handle rollback and/or commit for multiple distributed resources, but only at the cost of a great deal of complex, non-reusable logic.

E A R L Y D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

The most common solution to the problem of coordinating distributed transactions is to introduce a third participant, called a *transaction manager*, into the design. The transaction manager acts as a mediator between applications and the multiple resources the applications use.

7.1.5.1 The X/Open Distributed Transaction Processing Model

Enterprise database, messaging, and transaction monitor vendors have agreed on a standard set of interfaces for distributed transaction processing, called the *X/Open Distributed Transaction Processing Model* (see Figure 7.3), or DTP. The DTP model defines three participants in a distributed transaction: the transactional application and resource manager already shown in Figure 7.1, and the transaction manager, which coordinates the transactions of multiple resource managers, providing the application with ACID transactions across multiple resources. (A fourth DTP participant, a Communication Resource Manager, is beyond the scope of this chapter.)

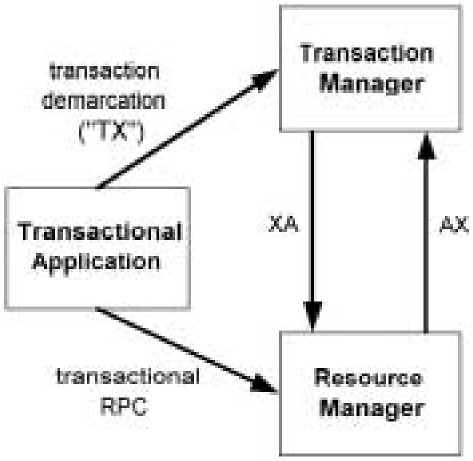

Figure 7.3 The X/Open standard reference model for distributed transactions

Standard interfaces decouple the distributed transaction participants from one another, providing interoperability. A transactional application demarcates transactions using the transaction manager's standard interface, called *TX*. The transaction manager manages transactions across multiple resource managers, each of which implements a standard interface called *XA*. The *AX* interface allows resource managers to register themselves with a transaction manager (but is not

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

required in J2EE technology). The application communicates with the resource manager using RPC as before, with the transaction managed implicitly by the transaction manager.

Application and component developers usually need to know only how to start, commit, and roll back transactions. The transaction manager transparently manages transactional access to any XA-enabled resource managers the application may use. J2EE Application Component Providers can use the Java Transaction Architecture (JTA) interface `UserTransaction` to communicate programmatically with the transaction manager (see "J2EE Platform Transactions," starting on page 190). Transaction demarcation for enterprise beans can be defined at deployment time; see "Container-Managed Transaction Demarcation," starting on page 201).

At any time during a distributed transaction, the transaction manager maintains an association between each transaction (which has a unique global ID), and connections to the resource managers used by that transaction. For example, a transaction manager may associate a single transaction ID with a SQL connection that has updated a table, a JMS provider waiting to transmit a message, and a Java Connector executing an external business function. The cumulative state of the association between the transaction and the resources it is using is called *transaction context*. Transaction context provides an application with a virtual transaction distributed across resource managers; coordination of this transaction context between the resource managers and the application is called *transaction context propagation*.

The next section describes the usual mechanism for controlling distributed transactions.

7.1.5.2 Two-phase Commit Protocol

Resource managers that don't "know" about one another can't cooperate directly in distributed transactions; instead, the transaction manager controls the transaction, indicating to each resource manager whether and when to commit or roll back, based on the global state of the transaction. A transaction manager coordinates transactions between resource managers using a *two-phase commit* proto-

E A R L Y   D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

col. The two-phase commit protocol provides the ACID properties of transactions across multiple resources..

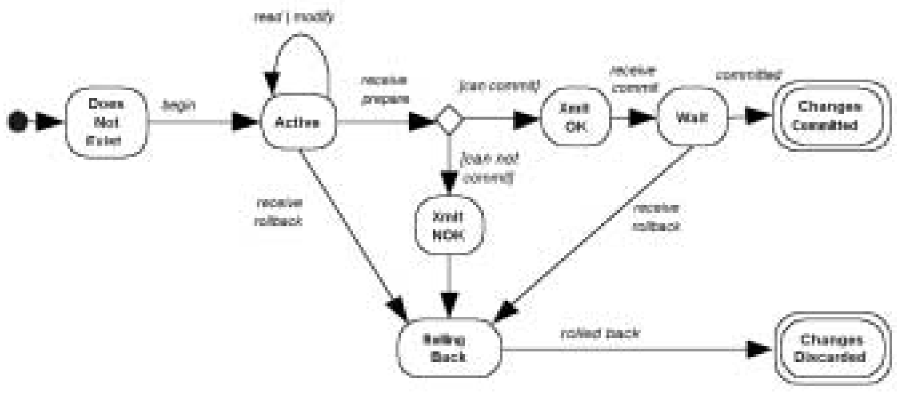

Figure 7.4 State transition diagram for a resource manager using two-phase commit

In the first phase of two-phase commit, the transaction manager tells each resource to “prepare” to commit; that is, to perform all operations for a commit, *except for* actually writing data to permanent storage. Each resource manager responds, indicating whether or not the prepare operation succeeded. In the second phase, if all prepare operations succeed, the transaction manager tells all resource managers to commit their changes; otherwise, it tells them all to roll back, and indicates transaction failure to the application.

Figure 7.4 shows a state diagram for a resource manager under the control of a transaction manager using two-phase commit. The nodes in the diagram indicate the state of a resource manager, and the edges indicate commands from the transaction manager, executed through the XA interface. Here’s how a two-phase transaction commit works, from the resource manager’s point of view:

- The first time a transactional application accesses a resource, the transaction manager tells the resource manager to *begin* the transaction (a process called “enlistment”). The resource manager enters an *active* state in which the application may read or modify data through the resource manager’s application interface.
- Remember that in the DTP model, applications demarcate transactions with the transaction manager, not with the resources themselves. When the application

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

tells the transaction manager to commit the transaction, the transaction manager tells each resource manager to *prepare* the transaction.

- If the prepare operation in a resource manager fails, the resource manager returns NOK (“not OK”) to the transaction manager and immediately rolls back its part of the transaction.
- If the prepare operation succeeds, the resource manager returns OK and enters a *wait* state, where it awaits the second phase. In the wait state, the resource manager can *only* commit or roll back: no additional resource reads or updates are permitted.
- If all other resource managers prepare successfully, each one receives *commit* from the transaction manager, and the resource commits its updates; otherwise, it will receive *rollback* from the manager, and will discard updates.

A particular resource manager may be involved in multiple simultaneous distributed transactions. The ACID properties apply for all of the resource managers involved in a particular distributed transaction, as well as for all of the concurrent partial transactions within a particular resource manager.

7.2 J2EE Platform Transactions

Enterprise applications require safe, reliable, recoverable data access, so support for transactions is an essential element of the J2EE architecture. The J2EE platform supports both programmatic and declarative transaction demarcation. The component provider can programmatically demarcate transaction boundaries in the component code with the Java^{TM} Transaction API (JTA). Enterprise beans support declarative transaction demarcation, in which the enterprise bean container automatically starts and completes transactions based on configuration information in the components’ deployment descriptor. In both cases, J2EE platform assumes the burden of implementing transaction management.

A J2EE Web or application server implements the necessary low-level transaction protocols, such as interactions between a transaction manager and JDBC database systems, transaction context propagation, and optionally distributed two-phase commit. Version 1.3 of the J2EE platform requires only support for so-called “flat” transactions, which cannot have any child (nested) transactions.

For Web applications, the J2EE platform supports a combination of servlets and/or JSP pages accessing multiple enterprise beans within a single transaction.

E A R L Y D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

Each component may acquire multiple connections to multiple shared resource managers.

J2EE transaction management is transparent to component and application code. From a J2EE component or application developer's point of view, the transaction manager is part of the server. A J2EE component developer can demarcate transactions programmatically using interface `javax.transaction.UserTransaction`, which is implemented by a server-side transaction manager. Or, a J2EE component deployer can declaratively control enterprise beans' transactional demarcation using the EJB server's deployment descriptor, in which case the EJB container controls transactions by interacting with the server-side transaction manger. See “How JTS and JTA support transactions” on page 197 for details.

An application can perform distributed transactions because a transaction manager propagates the transaction context across multiple resource managers. In addition, multiple applications may concurrently use a transaction manager. For example, a large retail store's point-of-sale, procurement, receiving, and inventory management applications may all access multiple overlapping transactional resources (such as an inventory database, and financials and ERP packages) by way of the J2EE application server's transaction manager. Transaction managers can also cooperate to propagate transaction context across server boundaries. In the retail store example above, the transaction manager for the store's application server may also interoperate with the transaction managers for other stores within the same enterprise, permitting customers to pay for an item at one location and have it delivered from another. Such transactional capabilities pave the way for a high level of integration across the enterprise and along the supply chain.

The next few sections provide examples of these transactional scenarios, which involve multiple distributed transaction participants.

7.2.1 Accessing Multiple Resources

As of version 1.3 of the J2EE platform, a J2EE product is required to support access within a single transaction to:

- a single JDBC database (multiple connections to the same database are allowed),
- a single Java Message Service (JMS) provider, and
- multiple Enterprise Information Systems (EISs) through a J2EE Connector Extension.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

See “J2EE Resource Manager Types,” starting on page 204 for more on these three types of resource managers.

Access to multiple JDBC databases within a single transaction is not required by J2EE version 1.3, and neither is support for multiple JMS providers within a transaction. Some Product Providers may add value to their product line by including these extra, non-standard transactional capabilities. For example, the J2EE SDK supports access to multiple JDBC databases in one transaction, mediated by a two-phase commit protocol.

Application designs often involve a tradeoff between functionality and portability; see the guideline “Avoid Unnecessary Portability Compromises,” starting on page 212, for an exploration of this topic.

Example: Transactions Across Multiple Resource Managers

The following scenario illustrates a J2EE transaction that spans multiple resources. In Figure 7.5, a client invokes a method on enterprise bean `X`. Bean `X` accesses database `A` using a JDBC connection. Then enterprise bean `X` calls a method on another enterprise bean `Y`, which sends a JMS message to some other system. The client then invokes a method on enterprise bean `Z`, which updates and returns some data from an external EIS system, using a Java Connector. The transaction manager in the J2EE server coordinates activities with the resource adapters for the database, the JMS provider, and the EIS system. The server ensures that the database update by bean `X`, the message transmission by bean `Y`, and the EIS operation performed by bean `Z` are either all committed, or all rolled back.

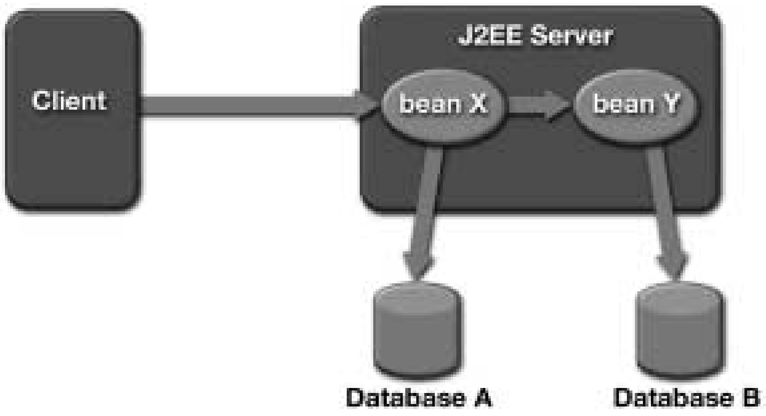

Figure 7.5 A Database, a JMS provider, and an EIS Within a Single Transaction

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

An Application Component Provider does not have to write extra code to ensure transactional semantics. Enterprise beans X, Y, and Z access their resources using the JDBC API, JMS, and a Java Connector, respectively. Behind the scenes, the J2EE server's transaction manager enlists the connections to all three systems as part of the transaction. When the transaction is committed, the J2EE server and the resource managers perform a two-phase commit protocol to ensure atomic update of the two systems.

7.2.2 Multiple Transactional Applications

More than one application can access a transaction manager concurrently. The transaction manager provides the isolation property for a distributed transaction, ensuring that transaction contexts corresponding to different applications (or distinct instances of the same application) do not interfere with one another. JTA transactional components include servlets, JSP pages, and enterprise beans. Product Providers may provide non-standard extensions that allow applets and/or application clients to participate in JTA transactions, as well.

For example, Figure 7.6 shows four applications in a large retail store accessing three separate resource managers transactionally, by way of a transaction manager in a J2EE server. Each distributed transaction is specific to a particular application. The transaction manager transparently propagates transaction context across the resource managers used by each transaction. It also enforces isolation between the data in each distributed transaction context.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

\

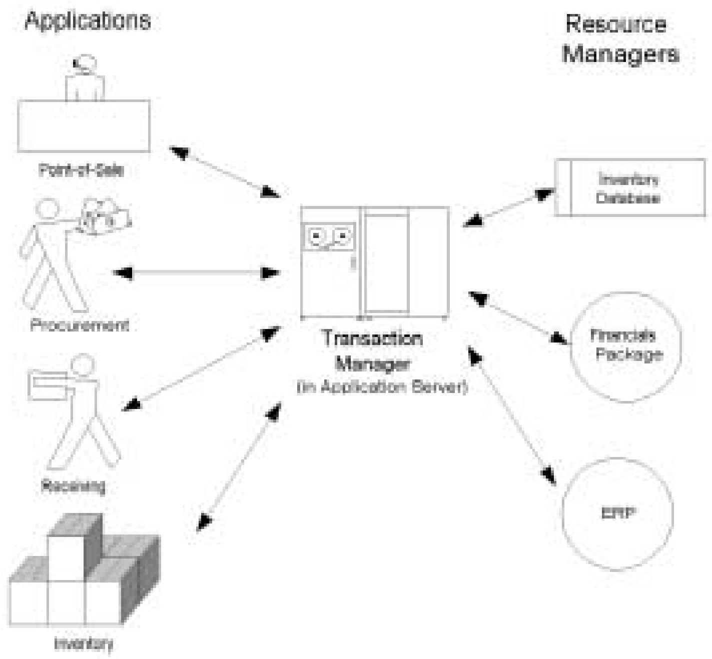

Figure 7.6 Multiple applications can perform concurrent distributed transactions

7.2.3 Transactions Across Servers

J2EE products can distribute transactions across multiple application servers.

Example: Transactions Across J2EE Servers

In Figure 7.7, a client invokes enterprise bean `X`, which updates data in enterprise information system `A`, and then calls another enterprise bean `Y` that is hosted by a dif-

E A R L Y   D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

ferent J2EE server. Enterprise bean Y performs read-write access to enterprise information system B.

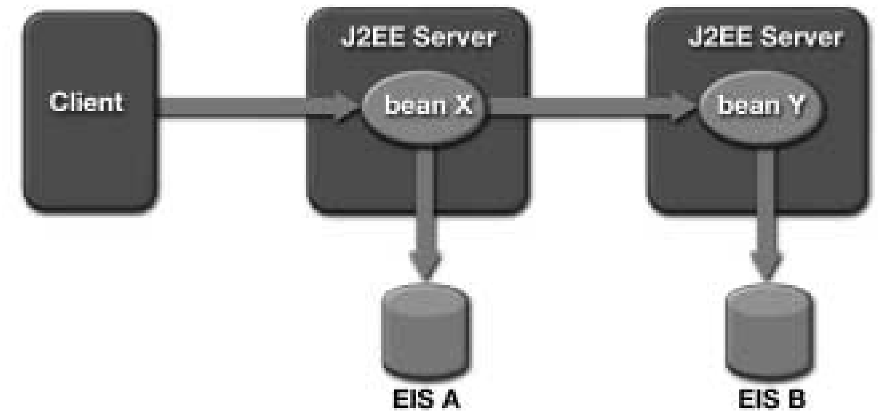

Figure 7.7 A transaction can span multiple J2EE servers

When X invokes Y, the two J2EE servers cooperate to propagate the transaction context from X to Y. This transaction context propagation is transparent to the application code. At transaction commit time, the two J2EE servers use a distributed two-phase commit protocol to ensure that the two enterprise information systems are updated under a single transaction.

7.3 J2EE Transaction Technologies

J2EE transactions are an object-oriented design for the DTP reference model. Two specifications define transactional behavior in the J2EE architecture. The Java Transaction Architecture (JTA) specification defines how applications, application servers, and resource managers communicate with a transaction manager, and define the transaction manager’s behavior. The Java Transaction Service (JTS) specification defines an implementation of a JTA transaction manager that can interoperate in a standard way with other JTA or non-JTA transactional objects.

Java Transaction Architecture (JTA)

JTA specifies standard Java interfaces between a transaction manager and the distributed transaction participants it coordinates: applications, application servers, and resource managers. JTA defines interfaces that let applications, application servers,

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

and resource managers participate in transactions regardless of their implementations.

A *JTA transaction* is a transaction managed and coordinated by the J2EE platform. A J2EE product is required to support JTA transactions as defined in the J2EE specification. A JTA transaction can span multiple components and enterprise information systems. They are propagated automatically between components and to enterprise information systems accessed by components within that transaction. For example, a JTA transaction may comprisea servlet or JSP page accessing multiple enterprise beans, some of which access one or more relational databases.

JTA transactions begin either explicitly in code, or implicitly by an EJB server. A component can explicitly begin a JTA transaction using the JTA interface `javax.transaction.UserTransaction`. An EJB container implicitly begins a JTA transaction when a client accesses an enterprise bean that uses container-managed transaction demarcation.

Various J2EE developer roles use different JTA interfaces. Application Component Providers can control transactions programmatically with JTA's interface `UserTransaction`. When using enterprise beans, Application Deployers can control transactions declaratively using the EJB container's deployment descriptor. JTA includes intefaces used by J2EE Product Providers, such as application server, resource manager, and resource adapter developers, and especially transaction manager developers.

Most J2EE Application Component Providers will only ever use the JTA `UserTransaction` interface, and perhaps not even that (if all transactions are managed by EJB containers). An Application Component Provider uses the JTA `UserTransaction` interface to demarcate JTA transaction boundaries in components. The JTS] `TransactionManager` and `XAResource` interfaces are low-level APIs between a J2EE server and enterprise information system resource managers and are not intended to be used by applications.

The main benefit of using JTA transactions is the ability to combine multiple components and enterprise information system accesses into one single transaction with little programming effort. (See "J2EE Transaction Technologies," starting on page 195.) The J2EE platform propagates transactions between multiple components and enterprise information systems with no additional programming effort. Enterprise beans using container-managed transaction demarcation can handle transactions with no code at all; they do not need to begin or commit transactions programmatically, because the EJB container automatically handles the demarcation.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

J2EE Blueprints recommends using JTA transactions to access EIS resources; see “Access Transactional Resources with JTA Where Possible,” starting on page 210.

Java Transaction Service (JTS)

JTS specifies the implementation of a JTA-compliant transaction manager, and provides distributed transaction interoperability with non-JTA transactional resources.

A JTS transaction manager implements the JTA interfaces, so it provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation in a J2EE application. While JTS implements JTA interfaces, JTS itself is not required by the J2EE 1.3 specification; specifically, EJB containers are not required to implement JTS interfaces.

A JTS transaction manager also implements the OMG Object Transaction Service (OTS) 1.1 specification, a low-level, language-neutral transaction service over IIOP. The OTS implementation allows JTS transaction managers to participate in distributed transactions with other transactional resources that implement OTS 1.1.

A J2EE platform implementation may implement JTS to support the transaction semantics defined in J2EE specification (as does the J2EE SDK, for example). The JTS implementation is transparent to J2EE components. Components never interact directly with JTS; instead, they demarcate transactions through the JTA `UserTransaction` interface.

JTS is primarily of interest to authors of transaction managers and application servers written in the Java language. Most J2EE application developers will have little or no direct contact with JTS APIs.

How JTS and JTA support transactions

J2EE component providers usually use only the JTA `UserTransaction` interface for programmatic transaction demarcation, or use an application server deployment descriptor for declarative transaction demarcation. Therefore, the following material is background information which, while not essential to use the technology, can help developers understand what’s going on “under the hood” of J2EE transaction management.

Figure 7.8 shows the participants and interfaces used in transactions in the J2EE architecture. The transaction manager and resource managers are the same

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

as in Figure 7.3, but the role of a transactional application in the DTP model can be played in the J2EE architecture either by an application or by an application server.

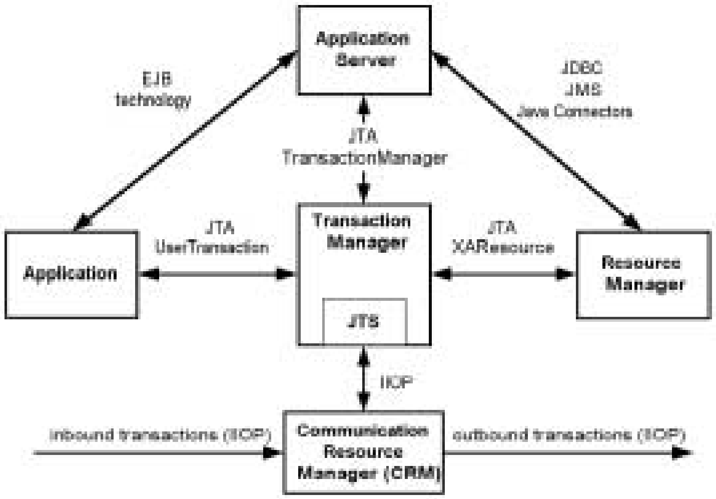

Figure 7.8 J2EE transaction architecture participants and interfaces

As shown in Figure 7.8, the JTA programming model defines the following interfaces between a transaction manager and its clients:

- Interface `javax.transaction.UserTransaction` is a transactional application's interface for transaction demarcation and other transaction manager services. An application may also access enterprise beans transactionally, in which case the enterprise bean container manages transactions automatically on behalf of the application, based on the contents of the bean's deployment descriptor.
- Interface `javax.transaction.TransactionManager` is how an application server requests services from a transaction manager. Its API is similar to that of `UserTransaction`, but also lets the application server suspend and resume transactions. If enterprise beans manage their own transactions, the EJB container forwards programmatic transaction demarcation requests to the transaction manager. If the enterprise bean container is managing transactions for the beans it contains, the container makes transaction demarcation requests when it determines they are necessary.
- Interface `javax.transaction.xa.XAResource` is how an XA-enabled resource

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

manager communicates with a transaction manager. `XAResource` is the Java mapping of the XA interface shown in Figure 7.3. The methods of interface `XAResource` implement the two-phase commit protocol described on page 188, as well as methods for propagating transaction context. Application servers can access resource managers transactionally by way of JDBC, JMS, or Java Connectors, depending on type of resource being accessed.

- JTS-compliant transaction managers participate in transactions with other OTS 1.1 transactional objects over IIOP. OTS 1.1 compliance lets a transaction manager propagate transaction context through CORBA interfaces to objects such as databases, or even to other transaction managers. The Communication Resource Manager makes the low-level transaction service protocol-independent.

7.4 Using Transactions in J2EE Applications

Availability of transactional facilities differ by application tier. This section discusses the transaction technologies available for each tier.

7.4.1 Client-tier Transactions

The J2EE platform does not require transaction support in applets and application clients, though like distributed transactions, a J2EE product might choose to provide this capability for added value. So, whether applets and application clients can directly access a `UserTransaction` object depends on the capabilities provided by the container. To ensure portability, applets and application clients should delegate transactional work to enterprise beans, either directly or by way of the Web tier.

7.4.2 Web-tier Transactions

A Web component in a two-tier application can access enterprise information systems under the scope of a JTA transaction. Web components support only programmatic transaction demarcation. A servlet or JSP page can use JNDI to look up a `UserTransaction` object (using the standard defined name `java:comp/UserTransaction`), then use the `UserTransaction` interface to demarcate transactions.

Code Example 7.1 illustrates the use of the JTA `UserTransaction` interface to demarcate transactions within a Web component:

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

```
Context ic = new InitialContext();
UserTransaction ut =
   (UserTransaction) ic.lookup("java:comp/UserTransaction");
ut.begin();
// access resources transactionally here
ut.commit();
```

Code Example 7.1 Web Component Using JTA Transactions

Calling `UserTransaction.begin()` associates the calling thread with a new transaction context. Subsequent accesses of transactional resources such as JDBC connections or Java Connectors implicity enlist those resources into the transaction. The call to `UserTransaction.commit()` commits the transaction, transparently engaging the two-phase commit protocol if necessary.

A Web component may start a transaction only in its `service` method. A transaction that is started by a servlet or JSP page must be completed before the `service` method returns; in other words, transactions may not span Web requests. If the service method returns with an open `UserTransaction` (that is, `begin()` has been called, but not `commit()` or `rollback()`), the container aborts the transaction and rolls back all data updates. (See the J2EE specification version 1.3, section J2EE.4.1.3.) For guidelines on using transactions with Web-tier J2EE technologies, see “Web-tier Transaction Guidelines,” starting on page 212.

7.4.3 EJB-tier Transactions

Enterprise beans offer two types of transaction demarcation: bean-managed and container-managed. In container-managed transaction demarcation, six different transaction attributes—`Required`, `RequiresNew`, `NotSupported`, `Supports`, `Mandatory`, and `Never`—can be associated with an enterprise bean method. An Application Component Provider or Assembler specifies the type of transaction demarcation and transaction attributes for the methods of the enterprise beans in the deployment descriptor.

This section discusses the types of transactions and the attributes of container-managed transactions and presents guidelines for choosing among the available options. It also describes the types of resource managers that the J2EE platform provides.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

7.4.3.1 Bean-Managed Transaction Demarcation

With bean-managed transaction demarcation, an enterprise bean uses the `javax.transaction.UserTransaction` interface to explicitly demarcate transaction boundaries. Session beans and message-driven beans can choose to use bean-managed demarcation; entity beans must always use container-managed transaction demarcation.

The following code illustrates the use of JTA interface to demarcate transactions in an enterprise bean with bean-managed transaction demarcation.

```
UserTransaction ut = ejbContext.getUserTransaction();
ut.begin();
// perform transactional work here
ut.commit();
```

Code Example 7.2 Enterprise Bean Using a JTA Transaction

The `UserTransaction` interface is used the same way in the EJB tier as in the Web tier, except that the reference to the interface is obtained by calling `EJBContext.getUserTransaction()`, instead of by way of a JNDI lookup. As noted in the section "Web-tier Transactions," resource managers are implicitly enlisted into a transaction, if one is active, the first time they are accessed from the thread that started the transaction. There is no way to explicitly pass the `UserTransaction` to a resource manager.

7.4.3.2 Container-Managed Transaction Demarcation

The EJB container manages transaction boundaries for an enterprise beans that use container-managed transaction demarcation. A transaction attribute for an enterprise bean method determines that method's transactional semantics, defining the behavior the EJB container must provide when the method is called. Transaction attributes are associated with enterprise bean methods in the bean's deployment descriptor. For example, if a method has a transaction attribute `RequiresNew`, the EJB container begin a new JTA transaction every time this method is called and attempt to commit the transaction before the method returns. The same transaction attribute can be specified for all the methods of an enterprise bean or different attributes can be specified for each method of a bean. Refer to Section 7.4.3.3 on page 202 for more information on transaction attributes.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

Even in container-managed demarcation, an enterprise bean has some control over the transaction. For example, an enterprise bean can choose to roll back a transaction started by the container using the method `setRollbackOnly` on the `SessionContext` or `EntityContext` object.

There are several benefits of using container-managed transaction demarcation:

- The transaction behavior of an enterprise bean is specified declaratively instead of programmatically. This frees the Application Component Provider from writing transaction demarcation code in the component.
- It is less error-prone because the container handles transaction demarcation automatically.
- It is easier to compose multiple enterprise beans to perform a certain task with specific transaction behavior. An Application Assembler that understands the application can customize the transaction attributes in the deployment descriptor without code modification.

7.4.3.3 Transaction Attributes

A *transaction attribute* is a value associated with a method of an enterprise bean that uses container-managed transaction demarcation. In most cases, all methods of an enterprise bean will have the same transaction attribute. For optimization purposes, it is possible to have different attributes for different methods. For example, an enterprise bean may have methods that don't need to be transactional.

A transaction attribute must be specified for the methods in the remote interface of a session bean and for the methods in the remote and home interfaces of an entity bean.

Required

If the transaction attribute is `Required`, the container ensures that the enterprise bean's method will always be invoked with a JTA transaction. If the calling client is associated with a JTA transaction, the enterprise bean method will be invoked in the same transaction context. However, if a client is not associated with a transaction, the container will automatically begin a new transaction and try to commit the transaction when the method completes.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

RequiresNew

If the transaction attribute is `RequiresNew`, the container always creates a new transaction before invoking the enterprise bean method and commits the transactions when the method returns. If the calling client is associated with a transaction context, the container suspends the association of the transaction context with the current thread before starting the new transaction. When the method and the transaction complete, the container resumes the suspended transaction.

NotSupported

If the transaction attribute is `NotSupported`, the transactional context of the calling client is not propagated to the enterprise bean. If a client calls with a transaction context, the container suspends the client's transaction association before invoking the enterprise bean's method. After the method completes, the container resumes the suspended transaction association.

Supports

It the transaction attribute is `Supports`, and the client is associated with a transaction context, the context is propagated to the enterprise bean method, similar to the way the container treats the `Required` case. If the client call is not associated with any transaction context, the container behaves similarly to the `NotSupported` case. The transaction context is not propagated to the enterprise bean method.

Mandatory

The transaction attribute `Mandatory` requires the container to invoke a bean's method in a client's transaction context. If the client is not associated with a transaction context when calling this method, the container throws `javax.transaction.TransactionRequiredException`. If the calling client has a transaction context, the case is treated as `Required` by the container.

Never

The transaction attribute `Never` requires that the enterprise bean method not be called within a transaction context. If the client calls with a transaction context, the container throws the `java.rmi.RemoteException`. If the client is not associated with any transaction context, the container invokes the method without initiating a transaction.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

J2EE Blueprints recommends using container-managed transaction demarcation where possible. For guidelines on where to use each transaction attribute in a design, see “Prefer Container-Managed Transaction Demarcation,” starting on page 210.

7.4.4 J2EE Resource Manager Types

J2EE architecture defines transactional behavior for three types of resource managers: JDBC-compliant databases, Java Connectors, and JMS providers. All three types of resource managers may be used within the scope of a single distributed transaction. Each type has somewhat different requirements and transactional semantics. This section describes these resource manager types and the requirements the J2EE platform specification places on them. It also explains their transactional semantics.

JDBC Resource Managers

A J2EE product is required to provide transactional access to at least one JDBC resource per transaction. Transactional access to JDBC resources is available from servlets, JSP pages, and enterprise beans. Multiple components accessing the same JDBC resource within the scope of the same transaction are supported. For example, a servlet may start a transaction, modify a database, and invoke methods on an enterprise bean that modifies that same database, all under the scope of the same transaction.

A commit operation on a JDBC resource manager means that any changes made under the scope of the transaction must be made permanent. Any subsequent operations that access the modified data will receive the new values. A rollback operation irrevocably discards any changes made to the data.

Transactional access to multiple JDBC resources, if available, is a non-standard extension.

JMS Providers

A J2EE product is required to support at least one JMS provider per transaction. Transaction access to a JMS provider is available from servlets, JSP pages, and enterprise beans. Like JDBC connections, multiple components, potentially in different tiers, must be able to access the JMS provider within the same transaction scope.

The transactional semantics of JMS message transmission refers to the transmission of the message itself, not necessarily any side-effect that transmission

E A R L Y D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

may cause. When a message is “sent” by a JMS provider under the scope of a transaction, the message is queued, but not actually transmitted. Transmission of messages queued during a transaction occurs only when the transaction is committed. If the purpose of the message is to perform some business function, failure of that function does not roll back the transaction. A rollback operation automatically dequeues and discards any JMS messages queued during that transaction.

Transactional access to multiple JMS providers, if available, is a non-standard extension.

Java Connector Extensions

Java Connector Extensions (“Java Connectors”) are resource adapters for specific EIS resources. Java Connectors are usually provided by providers of transactional resources to permit standards-based interoperation with J2EE products.

A J2EE product is required to support multiple Java Connectors that use `XA_TRANSACTION` mode. Transactional access from enterprise beans, servlets, and JSP pages is required, as is access from multiple components within the same transaction scope.

The resource manager for the Java Connector maintains the ACID properties of a transaction. Changes made to data due to invocations of Java Connector methods must be invisible to other transactions within that resource manager until the transaction is committed. When a transaction is rolled back, all effective data updates for that transaction must be discarded.

7.5 Transactions in Enterprise Information Systems

Most enterprise information systems support some form of transactions. For example, a typical JDBC database allows multiple SQL updates to be grouped in an atomic transaction.

Components should always access an enterprise information system under the scope of a transaction to guarantee the integrity and consistency of the underlying data. Such systems can be accessed under a JTA transaction or a resource manager (RM) local transaction.

7.5.1 JTA Transactions

When an enterprise information system is accessed under the scope of a JTA transaction, any updates performed on the system will commit or roll back depending on

E A R L Y D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

the outcome of the JTA transaction. Multiple connections to information systems can be opened and all updates through the connections will be atomic if they are performed under the scope of a JTA transaction. The J2EE server is responsible for coordinating and propagating transactions between the server and the enterprise information system.

If the J2EE product supports multiple enterprise information systems in one transaction, a J2EE application can access and perform updates on multiple enterprise information systems atomically, without extra programming effort, by grouping all updates within a JTA transaction. Code Example 7.3 illustrates this use:

```
InitialContext ic = new InitialContext("java:comp/env");
DataSource db1 = (DataSource) ic.lookup("OrdersDB");
DataSource db2 = (DataSource) ic.lookup("InventoryDB");
Connection con1 = db1.getConnection();
Connection con2 = db2.getConnection();

UserTransaction ut = ejbContext.getUserTransaction();
ut.begin();
// perform updates to OrdersDB using connection con1
// perform updates to InventoryDB using connection con2
ut.commit();
```

Code Example 7.3 Accessing Multiple Databases

7.5.2 Resource Manager Local Transactions

A *resource manager local transaction* (or *local transaction*) is a transaction specific to a particular enterprise information system connection. A local transaction is managed by the underlying enterprise information system resource manager. The J2EE platform usually does not have control of or knowledge about any local transactions begun by components. Access to a transactional enterprise information system is usually under a local transaction if no JTA transaction has been initiated. For example, if a servlet accesses a JDBC database without starting a JTA transaction, the database access will be under the scope of a local transaction, specific to the database.

Local transactions may also be used when the enterprise information system is not supported by the J2EE platform. For example, the J2EE platform specifica-

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

tion is not required to support object-oriented databases. As a result, a standard platform implementation cannot propagate any JTA transactions to an object-oriented database, and any access will be under local transactions.

JTA transactions are preferable to local transactions; see "Access Transactional Resources with JTA Where Possible," starting on page 210, for details.

7.5.3 Compensating Transactions

A *compensating transaction* is a transaction or a group of operations that undoes the effect of a previously committed transaction. A distributed transaction may include both JTA transactions and resource manager local transactions, but local transactions require explicit management logic. JTA-enabled resource managers handle rollback automatically by simply discarding any changes made since a transaction began. But each resource manager local transaction requires a compensating transaction that can undo the local transactions effects in case a rollback occurs.

Compensating transactions are useful if a component needs to access an enterprise information system that either does not support JTA transactions or is not supported by a particular J2EE product. In both cases, the enterprise information system will be accessed under the scope of a resource manager local transaction. Performing atomic operations on multiple EISs can be challenging when some of those systems do not participate in the JTA transaction. Compensating transactions meet this challenge by providing programmatic "rollback" of operations already committed under resource manager local transactions. Compensating transactions must be manually coded into application logic; the JTA API provides no standard way to handle them.

For example, suppose an application needs to perform an atomic operation that involves updating three enterprise information systems: two JDBC databases that supports JTA transactions and an enterprise resource planning system that does not. The application would need to define a compensating transaction for the update to the enterprise resource planning system. The approach is illustrated in Code Example 7.4.

```
updateERPSystem();
try {
    UserTransaction.begin();
    updateJDBCDatabaseOne();
    updateJDBCDatabaseTwo();
    UserTransaction.commit();
```

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

```
}
catch (RollbackException ex) {
    undoUpdateERPSystem();
}
```

Code Example 7.4 Compensating Transaction

The methods `updateERPSystem`, `updateJDBCDatabaseOne`, and `updateJDBCDatabaseTwo` contain code to access and perform work on enterprise information systems. The `undoUpdateERPSystem` method contains code to undo the effect of `updateERPSystem` if the JTA transaction does not commit successfully. If the ERP system in this example were accessible through the Java Connector API, it would also support JTA transactions, and the compensating transaction and its associated logic would be unnecessary.

For guidelines on implementing compensating transactions, see “Encapsulate Compensating Transactions as Session Beans,” starting on page 211.

Compensating transactions have a few pitfalls:

- *Committed transactions cannot always be undone*. Consider Code Example 7.4. If for some reason the method `undoUpdateERPSystem` fails, the data will be left in an inconsistent state.
- *Server crashes can compromise atomicity*. For example, if the system crashes immediately after the method `updateERPSystem`, the two database updates will not occur, resulting in a partial transaction.
- *Resource manager local transaction commits violate isolation*. When using compensating transactions with non-JTA resources, committed resource manager local transactions may subsequently be undone. In Code Example 7.4, a concurrent enterprise information system client might use data from the committed update to the enterprise resource planning system which might be rolled back later. In other words, updates committed by a resource manager local transaction are visible to other transactions, even though the distributed transaction has not yet committed.

An application that depends on compensating transactions must have extra logic to deal with potential failures and inconsistencies. The extra work and pitfalls of compensating transactions mean applications should avoid using them if

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

possible. Instead, use JTA transactions where possible to provide a simple and safe way to achieve the ACID properties across multiple components and enterprise information systems.

7.5.4 Isolation Level

An *isolation level* defines how concurrent transactions to an enterprise information system are isolated from one another. Enterprise information systems usually support the following the isolation levels:

- `ReadCommitted`: This level prevents a transaction from reading uncommitted changes from other transactions.
- `RepeatableRead`: This level prevents a transaction from reading uncommitted changes from other transactions. In addition, it ensures that reading the same data multiple times will receive the same value even if another transaction modifies the data.
- `Serializable`: This level prevents a transaction from reading uncommitted changes from other transactions and ensures that reading the same data multiple times will receive the same value even if another transaction modifies the data. In addition, it ensures that if a query retrieves a result set based on a predicate condition and another transaction inserts data that satisfy the predicate condition, re-execution of the query will return the same result set.

Isolation level and concurrency are closely related. The isolation level indicates the degree of responsibility given to the EIS for managing concurrent data access. A lower isolation level typically allows greater concurrency, at the expense of more complicated logic to deal with potential data inconsistencies. A higher isolation level typically allows simpler logic, at the expense of system performance due to internal EIS data locking to enforce ACID transaction properties. A useful guideline is to use the highest isolation level provided by enterprise information systems that gives acceptable performance.

For consistency, all enterprise information systems accessed by a J2EE application should use the same isolation level. The J2EE specification version 1.3 does not define a standard way to set isolation levels when an enterprise information system is accessed under JTA transactions. If a J2EE product does not provide a way to configure the isolation level, the enterprise information system

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

default isolation level will be used. For most relational databases, the default isolation level is `ReadCommitted`.

We recommend that you not change the isolation level within a transaction, especially if some work has already been done. Some enterprise information systems will force a commit if you attempt to change the isolation level.

7.6 J2EE Transaction Guideliness

This section presents guidelines for effective use of J2EE platform transaction services.

7.6.1 Access Transactional Resources with JTA Where Possible

J2EE BluePrints recommends accessing enterprise information systems, such as databases, within the scope of a JTA transaction. Transactional access guarantees data consistency and integrity, and ensures that work performed by multiple components through multiple enterprise information system connections is grouped as an atomic unit. It also groups as an atomic unit work performed on one or more independent enterprise information systems.

Where JTA transaction control is not possible, such as with resource managers that do not support JTA, consider using resource manager local transactions with compensating transactions; see “Compensating Transactions,” starting on page 207. Keep in mind that each enterprise information system accessed with local transactions will have to be committed or rolled back explicitly. In addition, components using local transactions need extra logic to deal with individual enterprise information system rollbacks or failures.

7.6.2 Prefer Container-Managed Transaction Demarcation

As mentioned previously, the recommended way to manage transactions is through container-managed demarcation. Declarative transaction management provides one of the major benefits of the J2EE platform, by freeing the Application Component Provider from the burden of managing transactions. Furthermore, the transaction characteristics of an application can be changed without code modification by switching the transaction attributes, making components useful in more contexts. Transaction demarcation should be selected with great care by someone who understands the application well. Bean-managed transaction demarcation is only for advanced users who want more control over the work flow.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

7.6.2.1 Transaction Attributes Guidelines

Most enterprise beans perform transactional work (for example, accessing a JDBC database). The default choice for a transaction attribute should be `Required`. Using this attribute ensures that the methods of an enterprise bean are invoked under a JTA transaction. In addition, enterprise beans with the `Required` transaction attribute can be easily composed to perform work under the scope of a single JTA transaction.

The `RequiresNew` transaction attribute is useful when the bean method needs to commit its results unconditionally, whether or not a transaction is already in progress. An example of this requirement is a bean method that performs logging. This bean method should be invoked with `RequiresNew` transaction attribute so that the logging records are created even if the calling client's transaction is rolled back.

The `NotSupported` transaction attribute can be used when the resource manager responsible for the transaction is not supported by the J2EE product. For example, if a bean method is invoking an operation on an enterprise resource planning system that is not integrated with the J2EE server, the server has no control over that system's transactions. In this case, it is best to set the transaction attribute of the bean to be `NotSupported` to clearly indicate that the enterprise resource planning system is not accessed within a JTA transaction.

We do not recommend using the transaction attribute `Supports.` An enterprise bean with this attribute would have transactional behavior that differed depending on whether the caller is associated with a transaction context, leading to possibly a violation of the ACID rules for transactions.

The transaction attributes `Mandatory` and `Never` can be used when it is necessary to verify the transaction association of the calling client. They reduce the composability of a component by putting constraints on the calling client's transaction context.

7.6.3 Encapsulate Compensating Transactions as Session Beans

Compensating transaction code should be encapsulated in a session enterprise bean with a bean-managed transaction. If the enterprise information system access logic is relatively simple, they can all reside in this bean; otherwise, the enterprise bean can invoke other enterprise beans to access the enterprise information system. If an enterprise bean's only responsibility is to access an enterprise information system that does not support JTA transactions, its transaction attribute should be set to `NotSupported`, to indicate that a JTA will not be used in the enterprise bean.

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

7.6.4 Avoid Unnecessary Portability Compromises

For application designers, portable solutions maximize the number of available future choices. Portability is essential for Component Providers, who rely on it for their products to be useful to their customers.

When using a particular J2EE product in an application design, application designers and developers should understand and distinguish between transaction capabilities that are required by the specification and capabilities that are optional extensions. Using only product features that are required by the specification maintains application portability between products. For example, if a J2EE application needs to access multiple databases under a single transaction, it will not run properly on a J2EE product that does not support multiple database access.

While portability is an important application design goal, it is not the only one. You may choose to consciously trade off some portability in order to leverage some non-standard feature of a particular J2EE product. In such cases, consider encapsulating nonportable extensions in a replaceable software layer. If you later need to port your application to a different platform, or if the optional feature becomes unsupported or operates differently in a subsequent release, the encapsulating interface will contain the changes and minimize impact on the rest of the system.

Some non-standard extensions impact portability more than others. When shopping for a J2EE products, look at several J2EE products and see how many support the optional feature you want. Commonly-available extensions tend to sacrifice portability less than uncommon ones. Since the interfaces to these features are by definition non-standard, though, the advice about encapsulating these features still applies.

In short, portability can be traded off for non-standard functionality, but choose those trade-offs consciously. Keep in mind that sacrificing portability ties your application to a particular vendor's product line.

7.6.5 Web-tier Transaction Guidelines

In a multitier environment, servlets and JSP pages's primary responsibilities are data presentation and user interaction, which are usually not transactional. Because transactions tend to be associated with business logic, database access

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

and other transactional work should be handled by transactional enterprise beans, instead of by JTA in the Web tier.

In designs that do not use enterprise beans, or where for some reason you choose to use Web-tier transactions, the following guidelines apply. JTA transactions, threads, and JDBC connections have many complex and subtle interactions. Web components should follow the guidelines stated in the transaction management chapter of the J2EE specification (version 1.3, section J2EE.4.1.4):

- JTA transactions must be started and completed only from the thread in which the `service` method is called. If the Web component creates additional threads for any purpose, these threads must not attempt to start JTA transactions. These additional threads will not be associated with any JTA transaction.
- JDBC connections acquired and released by threads other than the `service` method thread should not be shared between threads.
- JDBC `Connection` objects should not be stored in static fields.
- For Web components implementing `SingleThreadModel`, JDBC `Connection` objects may be stored in class instance fields. By definition, only one thread can ever access an instance of a Web component implementing `SingleThreadModel`; therefore, that instance can assume that fields that reference any JDBC connections, and those connections' transaction contexts, will not be shared with any other thread.
- For Web components not implementing `SingleThreadModel`, JDBC `Connection` objects should not be stored in class instance fields, and should be acquired and released within the same invocation of the `service` method.

7.7 Summary

This chapter provides the guidelines for using transactions on the J2EE platform. It describes the J2EE transactional model available to each J2EE component type—application clients, JSP pages and servlets, and enterprise beans—and enterprise information systems.

The J2EE platform provides powerful support for writing transactional applications. It contains the Java Transaction API, which allows applications to access transactions in a manner that is independent of specific implementations and a means for declaratively specifying the transactional needs of an application. These capabilities shift the burden of transaction management from J2EE Application

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

Component Providers to J2EE product vendors. Application Component Providers can thus focus on specifying the desired transaction behavior, and rely on a J2EE product to implement the behavior.

7.8 References and Resources

- For our latest thinking on transaction management, see

  `http://java.sun.com/j2ee/blueprints/transaction_management/`

- The following books are good standard references on transaction processing:
  - *Transaction Processing: Concepts and Techniques*
    Jim Gray, Andreas Reuter
    Published 1992 by Morgan Kaufman Publishers
    ISBN: 1558601902
  - *Principles of Transaction Processing for the Systems Professional*
    Philip A. Bernstein, Eric Newcomer (Contributor)
    Published 1996 by Morgan Kaufman Publishers
    ISBN: 1558604154

EARLY DRAFT

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

E A R L Y   D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

E A R L Y   D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm

E A R L Y   D R A F T

Contents © 1999-2001 by Sun Microsystems, Inc. All rights reserved.
This document is a draft produced for closed review—please do not redistribute.
Document last modified: November 10, 2001 11:01 pm