WebSphere® software

IBM

Jan Smolenski
Peter Kovari

Transactions in J2EE

This Redpaper discusses the general transaction concepts and fundamentals of J2EE applications. We describe a two-phase commit protocol and various distributed transaction processing (DTP) models where this protocol plays a fundamental role.

The paper is closely related and an extension to the IBM Redbook *Building Messaging-Based and Transactional Applications - Patterns for e-business Series*, SG24-6875.

© Copyright IBM Corp. 2003. All rights reserved.

Transaction concepts

A transaction in the business sense can be viewed as an activity between two or more parties that must be completed in its entirety with a mutually agreed-upon outcome. It usually involves operations on some shared resources and results in overall change of state affecting some or all of those resources. When an activity or a transaction has been started and the mutually agreed outcome cannot be achieved, all parties involved in a transaction should revert to the state they were in before its initiation. In other words, all operations should be undone as if they had never taken place.

There are many examples of business transactions. A common one involves transfer of money between bank accounts. In this scenario, a business transaction would be a two-step process involving subtraction (debit) from one account and addition (credit) to another account. Both operations are part of the same transaction and both must succeed in order to complete the transaction. If one of these operations fails, the account balances must be restored to their original states.

In the context of business software, we can express the above more precisely. A transaction is the execution of a set of related operations that must be completed together. This set of operations is referred to as a *unit-of-work.* A transaction is said to *commit* when it completes successfully. Otherwise it is said to *roll back*. In the example above, the money transfer operation is a unit-of-work, composed of debiting one account and crediting another.

Transactions can be encountered and discussed at many different levels, from high-level business transactions, such as the travel reservation request or a money transfer operation, to low-level technical transactions, such as a simple database update operation. Quite often, different sets of processing requirements are associated with these transaction levels. Usually, low-level technical transactions demand the most rigorous processing rules, which in addition to the ACID properties include a requirement of being synchronous and short-lived, while the requirements of high-level business transactions, especially in the B2B arena, are for practical reasons more relaxed. We focus on this after introducing some technical terminology.

The fundamental property of a transaction is its reliability. To achieve this, reliability transactions must possess four distinct characteristics: atomicity, consistency, isolation and durability. In technical terms this is collectively referred to as the ACID properties.

- **Atomicity:** Transaction must either execute completely or not at all. If every operation within the unit-of-work completes successfully, then transaction becomes committed, meaning the changes to the data are made permanent. Otherwise, the changes are undone (or rolled back) and transaction effects are nullified.
- **Consistency:** Data integrity and validity from a business standpoint must be maintained.
- **Isolation:** The effects of the transaction operations are not shared outside the transaction and the data being accessed is not affected by other processes until the transaction (unit-of-work) completes successfully. Transactions appear to run serially.
- **Durability:** Once a transaction successfully completes, the data changes become permanent and can survive a system failure.

The access to shared resources involved in a transaction is controlled by a resource manager. It is up to the resource manager to enforce and guarantee that the ACID properties of a transaction are maintained. Typically, relational database management systems (RDBMS), such as IBM DB2 and messaging-based systems, including Java Message Service (JMS) providers, such as IBM WebSphere MQ, can fulfill the roles of resource managers.

Resource managers provide APIs for application programmers to use. A program can initiate a new transaction by explicitly invoking a “begin” operation or implicitly when the first operation is requested. A transaction ends when a program invokes an explicit commit or a rollback operation or implicitly when a program terminates or a commit fails.

Distributed transactions

The transaction description provided so far assumed a simple case of one resource manager internally controlling all the resources involved in a transaction. Such transaction is commonly referred to as a *local transaction*. This picture gets more complicated if we expand the scope to include several resource managers and perhaps other cooperating transactional systems. Transaction extended in this manner is called a *distributed transaction*. The term *global transaction* is also used. This definition implies many processes, often on several machines cooperating together on behalf of a single distributed transaction. The common unit of work that they participate in is called a *global unit of work*. In the context of self-service applications, we will mostly deal with distributed transactions.

Distributed (or global) transactions must adhere to the same ACID properties as the non-distributed ones, but there is an inherent complexity revolving around the synchronization of the participants. Since the coordination of several resource managers is necessary in this case, a transaction manager external to the resource managers is required. We also need some kind of a “coordination” protocol that would be commonly understood and followed by all parties involved. This complexity is addressed by various distributed transaction processing (DTP) models.

A resource manager capable of participating in an externally coordinated transaction is called a *transactional resource manager*.

The transaction management products, such as IBM CICS, TXSeries/Encina, Tuxedo, and the latest J2EE-compliant application servers are among many examples of products supporting distributed transactions.

Flat and nested transactions

Most commercially available transaction management products support only a flat distributed transaction model. A flat transaction is a top-level transaction that cannot have any child transactions.

In contrast, a nested transaction allows for creation of a transaction embedded in an existing transaction. Such transaction is called a *child* or a sub-transaction, and the existing transaction is called a *parent* transaction. In this model, any level of nesting is allowed. A top-level transaction is one with no parent. A support of the nested transaction model among commercial software vendors is currently very limited.

Compensating transactions

In this context a compensating transaction is a transaction or a group of operations that undo the effects of a previously committed transaction. There are many circumstances where the compensating transactions may play a role:

- They may be used to restore consistency after an unrecoverable failure (system or communication) prevented a distributed transaction from normal completion. Transaction was left “in-doubt”. Some participants might have committed while others did not.
- When one of the global transaction participants is a non-transactional resource manager, it does not support the distributed transaction processing model (no two-phase commit support). We will see that this would be the case if the resource manager were not a JTA or XA compliant. If such a transaction performs a rollback, its non-transactional participant may need to be rolled back via the compensating transaction.

- Since a global transaction retains its ACID properties, the resources accessed in the transaction are not available and remain locked until the transaction is completed. In certain business transaction scenarios, especially ones that span several enterprise information systems (EIS), maintaining long-lived locks and restricting data access for extended periods of time may not be acceptable options. In these situations, it may be desirable not to map business transactions into global ones, but split them into more manageable units of work (global or local transactions) and provide compensating transactions to perform rollbacks.
- They may be used in some long-lived workflow type transactions (flows), sometimes called *sagas*, composed of several atomic transactions executing outside of the global unit of work, in the workflow controlled sequence. Since these individual transactions commit independently of each other, a failure in one of the downstream transactions may require some compensating transactions to reverse (undo) previously committed ones.

It is important to stress that an application that depends on compensating transactions must have extra logic to deal with the possible failures. Otherwise, the data may be left in an inconsistent state. There is also the potential of data updates being rolled back later. For these reasons, their usage should be carefully evaluated.

Extended transactions - the future

The distributed ACID transactions are best suited for relatively short-lived operations, to minimize locking and maximize concurrency tasks, executing in the reliable, closely coupled environment, due to the IIOP communication protocol used in many DTP models.

In contrast, the long-lived (long-running) operations in the loosely coupled environment (for example, some Web-based B2B scenarios) are usually poor candidates to carry the end-to-end ACID-compliant transactions. The solutions that have been implemented usually required the application-specific mechanisms (compensating transactions, ad-hoc locking, and so on), involving substantial programming efforts.

The ACID properties are sometimes too strong (unnecessarily restrictive) and certain business scenarios are better served if some of these properties are relaxed. Based on this premise and to deal with this issue, the concept of an extended transaction evolved.

An extended transaction is supposed to provide a flexible way of building transactional applications composed of transactions possessing different degrees of ACID properties and executing in a loosely coupled environment.

This work is currently under way. Its scope is best defined in the Java Specification Request - JSR 95 (J2EE Activity Service for Extended Transactions) and in the Web Services Specifications for Business Transactions and Process Automation. The latter consists of new specifications addressing transacted communications of Web Services, namely the Web Services Coordination (WS-Coordination) extensible framework specification, the Web Services Transaction (WS-Transaction) specification and a new language to describe business processes (Business Process Execution Language for Web Services, or BPEL4WS for short).

The JSR 95, on the other hand, builds upon the OMG Activity service framework for CORBA based middleware. This generic middleware framework allows for building many types of specific extended transaction (extended Unit of Work) models. The JSR 95 describes the system design and interfaces for a J2EE Activity service that is the specific realization, within the J2EE programming model, of the OMG Activity service.

> **Note:** IBM intends to make available Business Process Execution Language for Web Services (BPEL4WS) in future releases of WebSphere Application Server Enterprise V5.
>
> WebSphere Application Server Enterprise V5 provides the ActivitySession support to the J2EE components. The ActivitySession service provides an alternative unit of work scope to that provided by global transaction contexts. An ActivitySession context can encapsulate global transactions (see WebSphere Application Server Enterprise V5 documentation for more details).

For more information regarding the above topics, visit the following Web sites:

- JSR 95:

  `http://www.jcp.org/en/jsr/detail?id=95`

- OMG Activity service document:

  `http://www.omg.org/cgi-bin/doc?orbos/2001-11-08`

- Web Services Coordination:

  `http://www-106.ibm.com/developerworks/library/ws-coor/`

- Web Services Transaction:

  `http://www-106.ibm.com/developerworks/library/ws-transpec/`

Two-phase commit

The two-phase commit protocol has been widely adopted as the protocol of choice in the distributed transaction management environment. It is based on the Open System Interconnection (OSI/DTP) standard. This protocol guarantees that the work is either successfully completed by all its participants (resource managers and other transaction managers) or not performed at all. The data modifications are either committed together or rolled back together. The goal is to ensure that each participant in a global transaction takes the same action (everybody commits or everybody rolls back).

The transaction manager initiates the two-phase commit after all of the work of the transaction is complete and needs to be committed. The flow is as follows:

1. First phase: all participants are asked by the transaction manager to prepare to commit. If a given resource manager can commit its work, it replies affirmatively, agreeing to accept the outcome decided by the transaction manager. It can no longer unilaterally abort the transaction. Such resource manager is said to be in the ready-to-commit or prepared state. If a resource manager cannot commit, it responds negatively and rolls back its work (*unilateral rollback*).
2. Second phase: a transaction manager asks all resource managers to commit if all are in the ready-to-commit (prepared) state. Otherwise it requests to roll back. All resource managers commit or roll back as directed and return status to the transaction manager.

The simplified two-phase commit protocol flow in successful and aborted global transaction scenarios is graphically illustrated in Figure 1 on page 6.

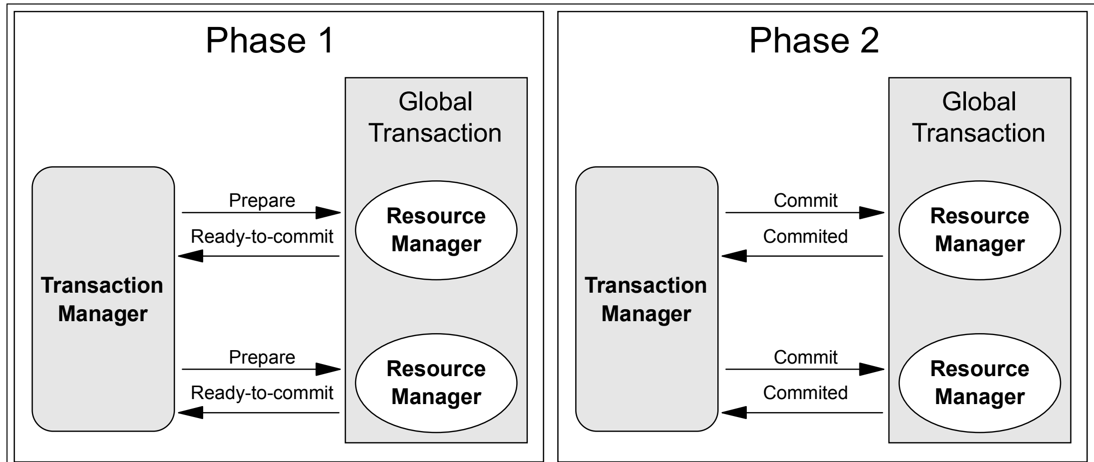


Figure 1 Successful two-phase commit illustration

Figure 1 illustrates the case when the transaction manager handles two resource managers in a two-phase commit transaction, where both resource managers are ready-to-commit and can commit at the end of the transaction.

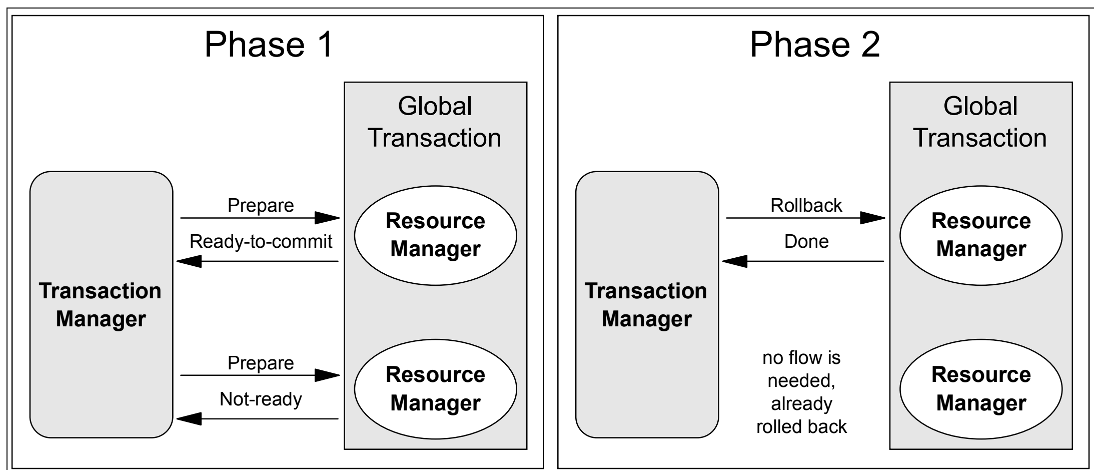


Figure 2 Aborted two-phase commit illustration

Figure 2 illustrates the case when the transaction manager handles two resource managers in a two-phase commit transaction, where one of the resource managers failed to prepare to commit the transaction.

It is important to stress the following:

- Any negative response or failure to respond within a timeout period to the phase 1 request vetoes (nullifies) the global transaction. All participants are asked to roll back.
- The phase 2 request is not sent to any resource manager that responded negatively (a resource manager rolls back on its own as noted in the phase 1 description).
- The two-phase commit mechanism uses a presumed rollback model. This means that participants must acknowledge the commit but not the rollback. If a failure occurs during a transaction, participants will attempt to roll back their portion of work.
- Read-only optimization takes place if the participant has only performed read operations. In this case it responds appropriately to the phase 1 request and the phase 2 exchange does not occur.

- One-phase commit optimization takes place if there is only one transaction participant (resource manager). In this case only phase 2 requests are made.
- The system/communication failures during the two-phase commit exchange may result in an inconsistent state where some managers commit while others don't. This is called a *window of doubt*. To resolve "in-doubt" transactions, special and often manual handling based on the information recorded in the transaction logs is necessary. Recovery may involve restoring resources from backups or applying compensating transactions.

Last Participant Support

Last Participant Support is an advanced transactional capability of the transaction manager to coordinate global transactions involving any number of two-phase commit capable resource managers and a single one-phase commit capable resource manager. When a global transaction involving such a mix of resources is about to be committed, the transaction manager uses the two-phase commit protocol to prepare all two-phase commit capable resources, and if this is successful, the one-phase commit resource is called to commit the transaction. Depending on the outcome of the one-phase resource commit operation, the two-phase commit resources are then committed or rolled back.

Note: WebSphere Application Server Enterprise V5 provides Last Participant Support advanced transactional capability.

Implicit transaction processing[1]

Now that we have a good understanding of transactions and messaging-based systems, let's demonstrate how these technologies could work together in the business application integration arena.

An excellent example of such an approach can be found in *The Five Axes of Business Application Integration*. This book provides an introduction to the practical aspects of business application integration. The transactions, messages, and process management (process and work flow automation) are considered to be among those five fundamental integration axes. Transaction and messaging are brought together in the concept of an implicit transaction. For more information, read the book or refer to the Web site at:

http://www.middlewarespectra.com

Implicit transaction processing is the primary basis for the integration of distributed applications. It uses messaging and depends on a series of assumptions that are acceptable (individually and in aggregate) to enable an organization to have sufficient confidence that its transactions (business and technical) will be processed accurately and completely. If such confidence cannot be established, and is needed, then explicit transaction processing will be considered.

An additional advantage of implicit transaction processing is that it can cater for both short-running transactions and long-running ones. While the short-running ones may not achieve the throughput that an OLTP (On-Line Transaction Processing) product does, the ability to handle long-running transactions (lasting hours or days or weeks) offers a broader range of options to the business. Just as a sleek sports car can move extremely fast for a short while and carry a minimal payload, so a truck can drive longer carrying much more. Implicit transaction processing is a broader tool that is applied generally and easily.

To understand the nature of the implicit transaction processing, see Figure 3 on page 8. This again illustrates the advantages of decoupling or asynchronous processing. System A has

[1] From *The Five Axes of Business Application Integration* by Charles Brett, published by Spectrum Reports Ltd., 2002 ISBN 0-9541518-1-X

Application A (say SAP's R/3 or a PeopleSoft application). This needs to communicate with Application B on System B (another application).

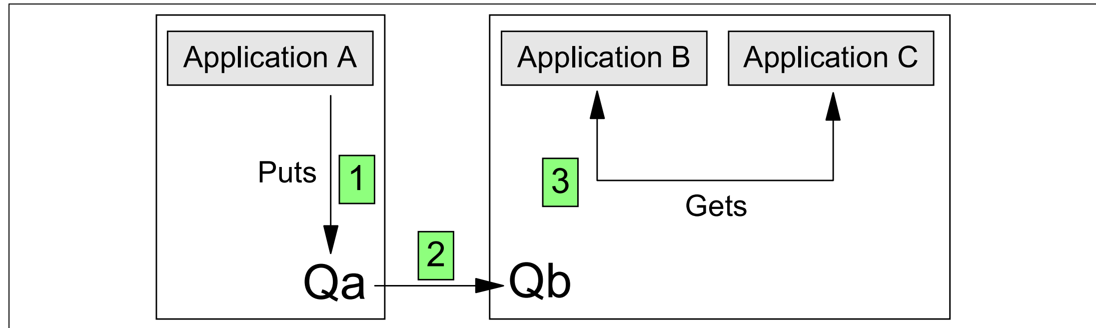


Figure 3 Implicit transaction processing

Traditionally the design and creation of an integration solution requires the implementation of some form of coordination manager, to handle how A talks to B and vice versa. This would likely be synchronous and might use explicit transaction management—meaning it would be expensive to build and run. Furthermore, if Systems A and B are different (say on a UNIX server or CICS on z/OS), the integration designer will have to understand both of these application platform environments and the communications between them, as well as their different formats, data representations (ASCII to EBCDIC), etc. On past evidence, building such a solution produces a monolithic design that is difficult to test, maintain, or change. Few organizations have the individuals with the breadth of skills to deliver this.

A decoupled approach with implicit transaction processing changes the solution. In Figure 3, there are five "decoupling elements":

1. Application A
2. Qa, the queue/queue manager on System A that communicates with other queues/queue managers via an agreed mechanism (whether this is via messaging or RPC or conversational does not matter to either Application A or B)
3. Qb, the queue/queue manager on System B
4. Application B
5. Application C

It is assumed that:

- The queues/queue managers (Qa and Qb) are able to communicate reliably with each other, for example, that they are using WebSphere MQ.
- Application A's development tools, and thereby each completed application, can place messages onto and read from Qa.
- Application B and C's development tools, and thereby each completed application, can read messages from as well as place them onto Qb.

Now, apply this to the processing of transactions between the applications (A, B and C) that are being integrated in Figure 3. Implicit transaction processing says that:

- **If** you can rely on Application A placing a message reliably on Qa (part of the developer's responsibility), (step [1] in Figure 3)
- **And** you can rely on a reliable transfer between a first platform and a subsequent one (between Systems A and B) (step [2] in Figure 3)

- **And** you can rely on the designated downstream Application (B or C) correctly reading the message on Qb (that originated from Application A) and then processing it (part of the developer's responsibility) (step [3] in Figure 3)
- **Then** you can implicitly rely on the business transaction, the "whole" transaction that occurs across applications A, B and C, as possessing transaction integrity

What should be clear is that implicit transaction processing is not as "perfect" as explicit transaction processing. On the other hand it provides a practical, middle-of-the-road way to integrate different applications in a heterogeneous distributed environment. In addition, because of such functions as messaging and queuing, it can work when not all required systems or resources are simultaneously available.

In turn, this means that long-running transactions are possible, although process management may be a preferable alternative for obtaining this. Messaging and queuing can hold messages for long periods, until the next applicable resource becomes available. This provides new dimensions for processing, ones that are not readily available with explicit transaction processing.

Short-running and long-running transactions[2]

The key difference between short- and long-running transactions lies less in the implementation of vendor technologies and more in the awareness of designers and developers.

Short-running transactions are applicable where you can be sure that all supporting resources are available (or are not available). Business transactions can be complex, such as "book me a vacation in the Caribbean when you know that you can obtain a flight under US$500 on a Thursday in May, plus a five-star hotel and a rental car that must be a Jeep unless a Cadillac is available". With several airlines, hotel chains, and rental car agencies to be queried and booked, this sort of problem is more susceptible to a long-running approach than a short-running one. After all, there are multiple enquiries and stages to be undertaken before the final reservations can be made.

Of course, you can divide all the booking enquiries into a plethora of smaller short-running ones. But this either requires you to have a human transaction manager or to build a master process to coordinate all the shorter processes. Almost always it is easier, when looking to deliver business integration, to adopt the implicit route.

Building blocks of transaction-based systems

As we have already indicated, distributed transactions are inherently more difficult to manage. They usually require a specialized system component to monitor transaction progress and coordinate transaction participants (such as resource managers), making sure that the global unit of work is committed or rolled back as needed and that the ACID-like reliability is maintained. The responsibility for transaction management and coordination in this environment lies within the transaction manager, also called the Transaction Processing (TP) Monitor. The master transaction manager that coordinates distributed transactions involving other transaction managers is called a *distributed transaction manager*.

Middleware

The term *middleware*, even in the narrow context of systems software, has many meanings and more or less precise definitions. In very general terms, it could be seen as a layer of software between separate applications (client and server for example) that provides extra

[2] From *The Five Axes of Business Application Integration* by Charles Brett, published by Spectrum Reports Ltd., 2002 ISBN 0-9541518-1-X

functionality and a common set of APIs to exploit this functionality. It serves as the "glue" connecting these applications.

Middleware type software or products generally fall into these classes:

- Message Oriented Middleware (MOM)
- Web (HTTP) servers
- Web application servers
- Transaction Processing Monitors (or transaction managers)
- Remote Procedure Calls (RPCs)
- Object Request Brokers (ORBs)
- Object middleware technologies
  - CORBA
  - Java-RMI
  - COM/DCOM
- Data connectivity

Having defined these classes, we then proceed to describe them in more details.

Middleware in general has the following characteristics, illustrating the variables that go into the choice of middleware for enterprise integration solutions:

- Number of applications:
  - How many applications will be connected together?
  - How interconnected do these applications need to be?
  - Does each application need to communicate with every other application?
  - Does each application only need to communicate with one other application, or some combination of the two?
  - How will disparate applications communicate in a consistent fashion?
- Routing rules:
  - Does each application always know what application(s) it needs to route data to statically?
  - Or is the routing of data determined dynamically? The routing is dependent upon some field in the data or some set of complex business rules.
  - Where are the routing rules defined and controlled?
  - Are they defined for each application?
  - Are they defined within or outside of each application?
- Data transformation:
  - How do disparate applications understand each other's native data formats?
  - Does each application expect data to be translated into its own format prior to being received?
  - Is each application expected to translate its data format to some agreed-upon data format before sending it to other applications?
  - Where does data transformation occur? Does it occur within the applications? Or outside of the applications?

- Scalability:
  - How much change is expected over time in a given environment?
  - Will systems be replaced? Added? Taken away?
  - How often does change occur?
- Operational considerations:
  - Are there processing components that make up the middleware?
  - Does each application handle its own inter-application communication?
  - Is a central or distributed management model preferred?

Each of these above characteristics and variable factors can be applied in determining the right architecture and middleware model given a specific set of requirements.

Message Oriented Middleware (MOM)

A very well-known MOM is IBM WebSphere MQ (formerly known as MQSeries). WebSphere MQ runs on many hardware platforms and supports most of the operating systems and communication protocols that are presently in use. It hides the complexities of the underlying platform and communication protocol by providing a set of four interoperable APIs:

- Message Queueing Interface (MQI)
- Java Message Service (JMS)
- Application Messaging Interface (AMI)
- Common Messaging Interface (CMI)

WebSphere MQ forms the basis for other products in the WebSphere MQ family:

- WebSphere MQ Integrator Broker
- WebSphere MQ Everyplace
- WebSphere Adapters
- MQSeries Workflow

More information about the WebSphere MQ family of products can be found at:

`http://www-3.ibm.com/software/integration/`

Web (HTTP) servers

The Web or HTTP server delivers the content to the Web client (usually a Web browser). It uses HTTP protocol for the client communication and it can deliver static Web content such as HTML pages. At the back end, it can communicate with the application server to invoke its services to access back-end data and/or generate the dynamic content.

Web application servers

See “Application servers” on page 12.

J2EE platform

The J2EE, in combination with J2SE, provides the following standard services related to our discussion:

- HTTP and HTTPS

  Defines client and server-side APIs for HTTP and HTTP over SSL protocols.

- RMI/IIOP

  Allows for RMI-style, protocol-independent programming. Includes support of CORBA IIOP protocol.

- Java IDL

  Allows applications to invoke external CORBA objects, using the IIOP protocol.
- JDBC API

  Allows for connectivity with relational database systems.
- Java Transaction API (JTA)

  Offers transactional support.
- Java Naming and Directory Interface (JNDI)

  Provides access to naming and directory services.
- Java Message Service (JMS)

  Supports point-to-point and publish/subscribe messaging models.
- JavaMail API

  Provides APIs and the service provider to support sending e-mail from applications.
- J2EE Connector architecture

  Provides pluggable resource adapters support for accessing Enterprise Information Systems, including connection, transaction, and security management.
- Java APIs for XML (JAXP)

  Provides support for SAX and DOM XML parsers and transform engines.
- Java Authentication and Authorization Service (JAAS)

  Allows for user authentication and extends user-based authorizations.

Application servers

Application servers are mid-tier servers in multi-tier system architectures that run business applications tying together disparate back-end data and integrating existing applications. They may provide additional functionalities such as transaction management, load balancing, or failover support. The application servers considered in this publication are enterprise-class application servers compliant with the open standards and specifications such as ones defined in the J2EE platform specification. At the moment they enjoy widespread business acceptance and the support of many vendors.

Web application servers are a class of application servers supporting Web-based applications and capable of interfacing Web (HTTP) servers, or containing embedded HTTP support, for the presentation layer.

Transaction managers

We now take a closer look at the DTP transactional models and provide the main characteristics of the transaction managers compliant with those models.

X/Open DTP model

The X/Open DTP model defines the XA interface, a bidirectional interface between a transaction manager and a resource manager. Transaction managers and resource managers use the two-phase commit protocol, described in “Two-phase commit” on page 5. All transaction/resource managers implementing XA interface according to the X/Open specification are said to be XA-compliant. Details of this specification are beyond the scope of this paper. It suffices to say that it is an open standard for coordinating changes to multiple resources, while ensuring the integrity of these changes.

The XA interface is used by transaction managers such as WebSphere MQ, TXSeries/Encina, and Tuxedo. Most of the commercial database systems (RDBMS) and messaging-based systems currently in use offer XA-compliant resource managers and therefore can participate in the distributed transactions.

OMG/OTS DTP model

The Object Management Group's Object Transaction Service (OMG/OTS or just OTS) specification is an extension of the X/Open DTP transactional model into the object-oriented and distributed objects arena. OTS retains interoperability with the X/Open model. It extends this model by providing transactional distributed object support for CORBA-based objects using the CORBA/IIOP communication protocol. We do not discuss this model further; instead we concentrate on the Java transactional model, which offers similar capabilities.

Java transaction model

The Java Transaction Service (JTS) and the Java Transaction API (JTA) specifications from Sun Microsystems, Inc. are the foundation of the Java transaction model for the Java 2 enterprise distributed computing environment. This is the key transaction technology for J2EE.

The JTS specifies the low-level transaction service implementation of a Java transaction manager. At a low level, JTS implements OTS functionality using Java bindings. At a high level, it supports the JTA specification.

The JTA API defines local Java interfaces required for the transaction manager to support distributed transaction management. This consists of three interfaces between a transaction manager and other DTP participants:

1. A high-level application transaction interface, implemented in the javax.transaction.UserTransaction interface. It defines methods for explicit transaction demarcation management by an application program.
2. A high-level application server interface, implemented in the javax.transaction.TransactionManager and javax.transaction.Transaction interfaces. It defines methods to manage the transaction and control the transaction boundaries.
3. The mappings of the X/Open XA interface, implemented in the javax.transaction.xa.XAResource interface. This interface is implemented in resource adapters such as a JDBC driver (for the relational database resource manager) or JMS provider (for the message queue server resource manager). With this implementation, XA-compliant resource managers can participate in JTA/JTS managed global transactions.

The JTA API is included in the Java 2 Enterprise Edition (J2EE) Standard Services specification.

Using the Java transactional model, an application (using JTA) can initiate global transactions or can participate in global transactions initiated by other JTS/OTS compliant transaction managers.

The term JTA transaction is used to denote a transaction managed and coordinated by the J2EE platform.

EJB model

The Enterprise JavaBeans architecture offers much more than the distributed transaction support. One of its main goals is to become a blueprint for creating distributed object-oriented business applications in the Java language. Here we are only concerned with the

transactional capabilities of this model. The full description of EJB architecture and its goals can be found in the Sun Microsystems Enterprise JavaBeans Specification, Version 2.0.

The EJB architecture supports distributed transactions (JTA transactions) by implementing the Java transaction model. It defines the EJB Server component as being responsible for the distributed transaction management and other lower-level system services. It also defines the EJB container as the component integrated with the EJB server whose purpose is to insulate the Enterprise JavaBeans (application code) from the complexities of an EJB server by providing simpler APIs for the Enterprise JavaBeans use.

The architecture does not define interfaces between the server and the container. It leaves it up to the vendor supplying these services.

In general, however, it is the EJB server's role to provide the transaction management (via JTS transaction manager) and the EJB container's role to provide transaction enablement and state management. The EJB container does that by providing JTA's javax.transaction.UserTransaction interface to its Enterprise JavaBeans. It must also provide the JDBC and JMS support as well as the Connector APIs.

We cover JTA and Connector APIs in the context of EJB transaction management later in this paper. We will not cover JTS in any more detail, since this is a low-level interface not "visible" at the application level.

Note: The EJB architecture currently supports only flat transactions.

Designing transactions

Earlier, we discussed the definition of a transaction and defined the transaction properties, commonly known as ACID properties. The level of reliability that these properties guarantee makes transactions fundamental entities (building blocks) of any serious business solution (ranging from very simple to very complex).

The transactions involving a single resource manager (RM) are the most common. Any business application accessing a relational database system (RDBMS), for example, involves an ACID transaction that is internally controlled by the RDBMS resource manager. These internal resource manager controlled transactions are very efficient, although the efficiency could be compromised with poorly designed database schemas or poor database programming practices.

The transaction manager (TM) controlled local transactions involving only one resource manager are also quite efficient. These types of transactions are optimized to use the one-phase commit coordination protocol. This protocol flows only the "confirm-confirmed" or "commit-committed" type of request and reply between the transaction and the resource managers. In some cases, the resource manager can only support one-phase commit, so-called sync-level 1 RMs, so there is no choice. In other cases, when the two-phase commit capable RM (for example, an XA-compliant resource manager) is involved, the transaction manager performs the optimization by utilizing the more efficient one-phase commit protocol.

The distributed transactions involving multiple XA-compliant resource managers and especially those where the distributed transaction manager is used (the distributed transaction manager is the manager coordinating other transaction managers) are very costly. The main reason lies in the complexity of the two-phase commit protocol itself and the fact that in most cases the protocol flows over the network to the remotely located resource and transaction managers.

Preferably, all ACID transactions should be short-lived. The resource locking for an extended period of time is not acceptable because it limits concurrency. The longer the execution time, the more probable are transaction failures and the more difficult the recovery. Putting all these facts together leads us to the conclusion that reliance on the distributed transactions should be carefully evaluated every time they come into play in designing robust applications. Alternative approaches, such as one-phase commit and usage of compensating transactions, should be taken into consideration.

Building well-designed, reliable, scalable, and robust transaction-based business solutions (applications) is not an easy undertaking. Lots of factors need to be considered and the best design and implementation practices have to be put in place.

Fortunately, J2EE platform provides a lot of help in such an endeavor. J2EE-compliant application servers provide an embedded distributed transaction support to the EJB-based or Web-based components, within the EJB container and the Web container respectively. They take care of most of the low-level transaction management-related issues that we have already discussed (see “Building blocks of transaction-based systems” on page 9) and of such issues as concurrency, security, persistence, and scalability. All this, plus the richness of Java-based interfaces (APIs) offered by J2EE, greatly simplifies the development process of the transactional and distributed applications.

J2EE 1.3 transactional support considerations

The J2EE 1.3 platform offers the following kinds of transaction design options for its Enterprise Java Bean and Web components:

- EJB entity beans
  - Container-managed transactions
- EJB session beans and message-driven beans
  - Container-managed transactions
  - Bean-managed transactions
    - JTA transactions
    - JDBC transactions
- Web components (Servlet, JSP)
  - JTA transactions
  - JDBC transactions

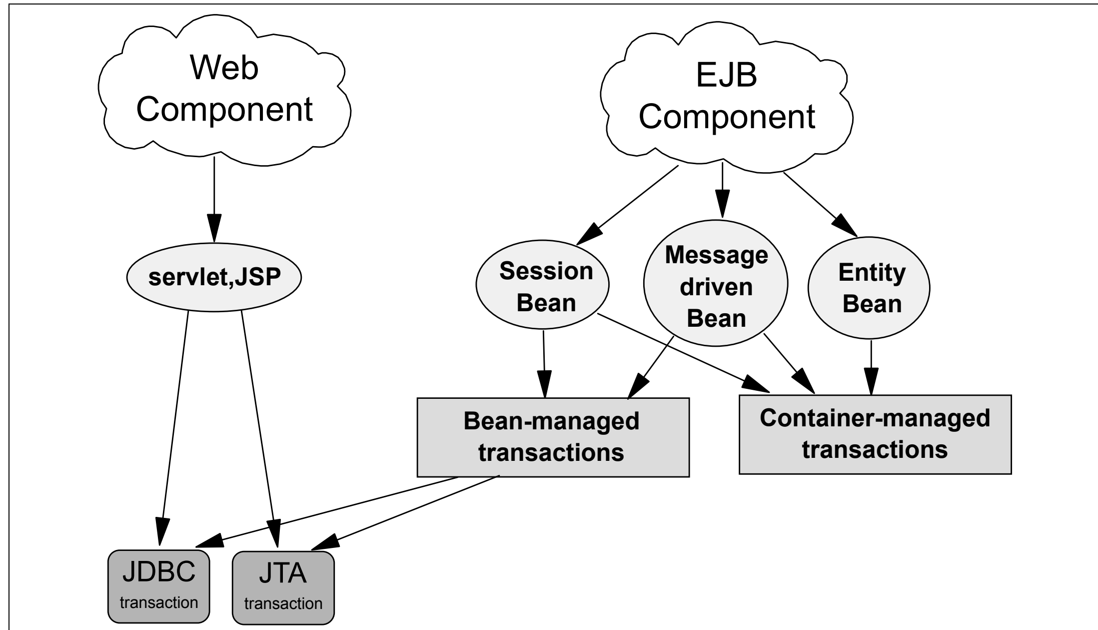


Figure 4 J2EE 1.3 components transaction options

As we can see, the J2EE transaction service provided by the EJB container supports two types of transactions in EJB-based applications: container-managed and bean-managed transactions. The transaction-type element of the session bean or message-driven bean descriptor determines if the container-managed or the bean-managed transaction demarcation is used. The transaction-type is not supported for the entity beans, because they are always container-managed.

EJB container-managed transaction (CMT) considerations

In container-managed transactions, the EJB container manages the transaction demarcation. Container-managed transactions include all entity beans and any session or message-driven beans with a transaction type set to container.

If the EJB-based application requires transactional support, then the container-managed transaction demarcation is the easiest and the recommended approach. Apart from being supported by all types of EJB beans, it additionally offers the following advantages:

- It is controlled declaratively, outside of the bean code, using the transaction attributes and the EJB deployment descriptor.
- It is set at the bean (or bean method) level and forces method invocation to be either completely included or excluded from a transaction (unit of work).
- It simplifies programming, because there is no code to begin and end the transaction. Transactions do not need to be explicitly started and terminated.
- When used in the message-driven bean, it offers the only way to make the receipt of the initial message (that is, an MDB triggering message) to be part of a transaction.

  To achieve this, the transaction attribute of Supported must be specified.
- It is the only choice for the entity beans.
- It offers a way to suspend an existing transaction and initiate a new one when a method is invoked.

  To achieve this, the transaction attribute of RequiresNew must be associated with the invoked method.

Important: Care must be taken not to process the JMS Request/Reply type of scenario within the same transaction scope. The message is not delivered until a transaction is committed. Therefore, the synchronous receipt of a reply to such a message within the same transaction will not be possible.

Transaction attributes

Transaction attributes determine how transactions are managed and demarcated.

The transaction attributes can be specified at the bean or at the method level. If present in both places, the method attribute will take precedence. The transaction attributes (EJB 2.0) are as follows:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Table 1 on page 18 summarizes the container's actions for each of the above transaction attributes. As we can see, this behavior depends on the existing transactional context (transaction scope).

In this table, T1 represents the transaction context of the client, possibly another EJB bean, that is invoking (calling) a container-managed bean method of another bean. T2 is the new transaction context that may be created when such a method is invoked. As we can see, the transaction attribute of the bean method, together with the transaction context of the client, influence the container action.

Note: Message-driven beans can only support the Required or NotSupported transaction attributes (see Table 1). This is a logical consequence of the fact that MDB is asynchronously driven by the container and cannot inherit any transaction context from the client nor throw an exception for the client to handle.

Tip: The only way to make the triggering JMS message part of a transaction is to use the container-managed transaction for the MDB and specify the transaction attribute of Required. In this case, the container-initiated transaction includes the receipt of a message and the onMessage method in the same transaction scope. When onMessage successfully completes, the container commits the transaction and the JMS message is permanently removed from the queue. In the case of an onMessage failure or when rollback is requested, the container performs the rollback and JMS message is re-delivered.

Table 1 Container-managed transaction demarcation rules

Transaction attribute	Invoking client's transaction context	Invoked bean's method transaction context	Comments
Required	none	T2	Starts T2 context
	T1	T1	Same context (T1) continues
RequiresNew	none	T2	Starts T2 context
	T1	T2	T1 is suspended and T2 is started
Mandatory	none	error	Throws error - T1 context required
	T1	T1	Same context (T1) continues
NotSupported	none	none	
	T1	none	Runs outside of T1 context
Supports	none	none	
	T1	T1	Same context (T1) continues
Never	none	none	
	T1	error	Throws error - cannot run in T1

Design/programming considerations

Important design and programming considerations for container-managed transactions are as follows:

- A container-managed transaction is automatically committed when the method successfully completes.
- A container-managed transaction is rolled back in one of these ways:
  - When a Java system exception is thrown.
  - When the transaction is marked for rollback with the setRollbackOnly method call of the javax.ejb.EJBContext interface. This action tells the container to roll back the transaction as soon as the method returns. It allows such methods to "gracefully" return with the return value or the application exception after the rollback has been performed.

  > **Tip:** The setRollbackOnly call can be used with the Required, RequiresNew, or Mandatory transaction attributes only, because they guarantee transactional context. Otherwise, an exception is thrown.

- Stateful session beans may implement the SessionSynchronization interface, so that they are notified by the container when certain transactional events are taking place. The implementation of this interface is optional. It is used to synchronize this session bean state, the instance variables, with the state of the transaction or, in other words, to maintain the transactional conversation. The following interface methods are available:
  - afterBegin

    Informs a session bean instance that a new transaction has just started. This method executes within the transaction context.

– beforeCompletion

  Informs a session bean instance that a transaction is about to be committed. This method executes within the transaction context.

– afterCompletion

  Informs a session bean instance that a transaction has just completed. It is invoked with a boolean parameter indicating that it was committed (value of true) or rolled back (value of false). This method executes outside of the transaction context.

Example 1 afterCompletion() method implementation

```
public void afterCompletion(boolean committed) {
   if(committed == false) {
      //transaction completed with a rollback:
      //restore previous state
      try{
         currState = oldState;
         ...
      }catch(Exception e){
         throw new EJBException("afterCompletion failure: " + e.getMessage());
      }
   }
}
```

Transaction semantics for container-managed transactions

Table 2 through Table 5 on page 21 describe the transaction semantics for EJB beans (stateful session beans, stateless session beans, entity beans and message-driven beans) in container-managed transactions.

> **Note:** The term *unspecified transaction context* is used in the EJB specification to refer to the cases in which the EJB architecture does not fully define the transaction semantics of an enterprise bean method. In practice, this means that to build portable EJBs, the EJB bean must be written to avoid relying on any particular container behavior or make any assumptions regarding the transaction context of such methods.
>
> In specific (J2EE 1.3 compliant) application server implementations, such as WebSphere Application Server V5, this behavior (that is, transaction semantics) in an unspecified transaction context is fully defined. An EJB can be optimized to take advantage of such specific implementation. In WebSphere Application Server V5 a local transaction containment (LTC) is used to define the application server behavior in an unspecified transaction context. The container always establishes an LTC before dispatching a method on an EJB or Web component, whenever the dispatch occurs in the absence of a global transaction context. For example, an EJB deployed with NotSupported or Never transaction attribute runs in an unspecified transaction context (per EJB specification), but in an LTC context under the WebSphere Application Server V5 (see the WebSphere Application Server V5 documentation for more details).

Table 2 Container-managed transaction semantics in the stateful session bean

EJB method	Transaction context at method invocation time
newInstance (Constructor)	Unspecified
setSessionContext	Unspecified
ejbCreate	Unspecified
ejbRemove	Unspecified

EJB method	Transaction context at method invocation time
ejbActivate	Unspecified
ejbPassivate	Unspecified
business methods	Determined by the transaction attribute (Unspecified if NotSupported or Never or Supports transaction attribute is used)
afterBegin	Yes - determined by the transaction attribute of the business method
beforeCompletion	Yes - determined by the transaction attribute of the business method
afterCompletion	No

Table 3 Container-managed transaction semantics in the stateless session bean

EJB method	Transaction context at method invocation time
newInstance (Constructor)	Unspecified
setSessionContext	Unspecified
ejbCreate	Unspecified
ejbRemove	Unspecified
business method	Determined by the transaction attribute (Unspecified if NotSupported or Never or Supports transaction attribute is used)

Table 4 Container-managed transaction semantics in the entity bean

EJB method	Transaction context at method invocation time
newInstance (Constructor)	Unspecified
setEntityContext	Unspecified
unsetEntityContext	Unspecified
ejbCreate	Determined by the transaction attribute of the matching create method (that is, one that triggered ejbCreate)
ejbPostCreate	Same as in the previous ejbCreate method
ejbRemove	Determined by the transaction attribute of the matching remove method (that is, one that triggered ejbRemove)
ejbActivate	Unspecified
ejbPassivate	Unspecified
ejbLoad	Determined by the transaction attribute of the business method that triggered the ejbLoad method
ejbStore	Same as in the previous ejbLoad method
ejbHome	Determined by the transaction attribute of the matching home method
ejbSelect	Determined by the transaction attribute of the invoking business method
business method	Determined by the transaction attribute (Unspecified if NotSupported or Never or Supports transaction attribute is used)

Table 5 Container-managed transaction semantics in the message-driven bean

EJB method	Transaction context at method invocation time
newInstance	Unspecified
setMessageDrivenContext	Unspecified
ejbCreate	Unspecified
ejbRemove	Unspecified
onMessage	Determined by the transaction attribute (either Required or NotSupported must be used)

EJB bean-managed transaction (BMT) considerations

In bean-managed transactions, the code in the session or message-driven bean explicitly controls the transaction demarcation. Bean-managed transactions include any session or message-driven beans with a transaction-type set to Bean.

Although the container-managed transactions are usually recommended for the reasons outlined in the previous section, the bean-managed transaction demarcation has its place, too.

In general, it offers an alternative, a programmatic rather than declarative approach to the transaction demarcation control. However, there are also some potential advantages. The main features of this approach are as follows:

- It is controlled programmatically from within the bean method code by issuing explicit Java javax.transaction.UserTransaction interface method calls.
- It offers “finer” granularity of transaction control than the container-managed ones, at a Java statement level rather than at a method level, which could be beneficial in the following scenarios:
  - It may be desirable to have a method where a transaction is conditionally initiated or where only part of a method requires transactional protection (scope).
  - It may be desirable to initiate transaction in one method and commit it in another method, which works only with the stateful session beans.
- It offers the ability to use JDBC type or JTA type transactions.

Design and programming considerations

Important design and programming considerations of the bean-managed transactions are as follows:

- Bean-managed transaction is only supported for the session bean and the message-driven bean. Entity beans can only use container-managed transaction.
- A stateless session bean instance method that started a transaction must complete the transaction before returning. If the transaction is not explicitly committed, it will be rolled back by the container and the EJBException or the RemoteException will be thrown.
- A message-driven bean instance onMessage method that started a transaction must complete the transaction before returning, which is the same behavior as the stateless session bean methods. If the transaction is not explicitly committed, it will be rolled back by the container.
- A stateful session bean instance business method that started the transaction does not have to complete this transaction before returning. The same transaction can continue across many client-bean interactions. The same transaction context will be associated

with all subsequent instance method invocations until the instance completes the transaction.

- A JTA transaction is programmatically controlled with the javax.transaction.UserTransaction interface method invocations, as follows:
  - A *begin* method call starts a transaction (possibly global).
  - A *commit* method call to commit (complete) the transaction.
  - A *rollback* method call to roll back changes and end the transaction.
  - A *setTransactionTimeout* method call. When the timeout expires while transaction is still in progress, it will be rolled back.
  - A *setRollbackOnly* method modifies the transaction associated with the target object such that the only possible outcome of the transaction is to roll back the transaction.

Example 2 Bean-managed JTA transaction

```
public void processOrder(String orderMessage) {
   UserTransaction transaction = mySessionContext.getUserTransaction();
   try{
      transaction.begin();
      orderNo = sendOrder(orderMessage);
      updateOrderStatus(orderNo, “order sent”);
      transaction.commit();
   }catch(Exception e){
      try{
         transaction.rollback();
      }catch(SystemException se){
         se.printStackTrace();
      }
      throw new EJBException(“Transaction rolled back: “ + e.getMessage());
   }
}
```

- A JDBC transaction is programmatically controlled with the java.sql.Connection interface method invocations, as follows:
  - The first SQL statement after connect, commit, or rollback implicitly initiates a JDBC transaction.
  - A *setAutoCommit(false)* prevents resource manager from implicitly committing every SQL statement.
  - A *commit* method call to commit (complete) the transaction.
  - A *rollback* method call to roll back changes and end the transaction.

Example 3 Bean-managed JDBC transaction

```
public void processOrder(String order) {
   Context initCtx = new InitialContext();
   javax.sql.DataSource ds = javax.sql.DataSource)initCtx.lookup
(“java:comp/env/jdbc/OrderDB”);
   java.sql.Connection conn = ds.getConnection();
   try{
      conn.setAutoCommit( false );
      orderNo = createOrder( order );
      updateOrderStatus(orderNo, “order created”);
      conn.commit();
   }catch( Exception e ){
      try{
         conn.rollback();
```

```
            throw new EJBException("Transaction rolled back: " + e.getMessage());
        }catch( SQLException sqle ){
            throw new EJBException("Rollback SQL error = " + sqle.getMessage());
        }
    }
}
```

Mixing JTA and JDBC transactions is not recommended.

Transaction semantics for bean-managed transactions

Table 6 through Table 8 on page 24 describe the transaction semantics for Enterprise JavaBeans (stateful session beans, stateless session beans, and message-driven beans) in bean-managed transactions.

Note: The term *unspecified transaction context* is used in the EJB specification to refer to cases in which the EJB architecture does not fully define the transaction semantics of an enterprise bean method. In practice, this means that the EJB bean must be written to avoid relying on any particular container behavior or making any assumptions regarding the transaction context of such methods.

In specific (J2EE 1.3 compliant) application server implementations, such as WebSphere Application Server V5, this behavior (that is, transaction semantics) in an unspecified transaction context is fully defined. An EJB can be optimized to take advantage of such specific implementations. In WebSphere Application Server V5, a local transaction containment (LTC) is used to define the application server behavior in an unspecified transaction context. The container always establishes an LTC before dispatching a method on an EJB or Web component, whenever the dispatch occurs in the absence of a global transaction context (see the WebSphere Application Server V5 documentation for more details).

Table 6 Bean-managed transaction semantics in the stateful session bean

EJB Method	Transaction context at the method invocation time	Can use UserTransaction interface methods?
newInstance (Constructor)	Unspecified	No
ejbCreate	Unspecified	Yes
ejbRemove	Unspecified	Yes
ejbActivate	Unspecified	Yes
ejbPassivate	Unspecified	Yes
setSessionContext	Unspecified	No
business method	If the previous method returned without completing the transaction, then the transaction context is inherited. Otherwise No.	Yes

Table 7 Bean-managed transaction semantics in the stateless session bean

EJB Method	Transaction context at the method invocation	Can use UserTransaction interface methods?
newInstance (Constructor)	Unspecified	No
ejbCreate	Unspecified	Yes
ejbRemove	Unspecified	Yes
setSessionContext	Unspecified	No
business method	No	Yes

Table 8 Bean-managed transaction semantics in the message-driven bean

EJB Method	Transaction context at the method invocation time	Can use UserTransaction interface methods?
newInstance (Constructor)	Unspecified	No
setMessageDrivenContext	Unspecified	No
ejbCreate	Unspecified	Yes
ejbRemove	Unspecified	Yes
onMessage	No	Yes

Web component transaction considerations

The Web components (servlets and JSPs) transaction management considerations are similar to the bean-managed transactions. Both JTA and JDBC type transactions are supported. The transaction initiated in the servlet's service method must be completed (committed or rolled back) in the same method.

Message flows

The message broker functionality introduced the concept of message flows, which could be seen as yet another transactional model complementing the distributed transaction processing model discussed so far. It offers extra capabilities, especially in the seamless application integration arena.

A message flow represents a sequence of operations to be executed based on the retrieved message. The message broker retrieves a message from the message queue and acts upon it according to the predefined message flow. Typically, the message flow encapsulates multiple processing steps, such as data filtering, transformation, and dynamic routing. The message is propagated from step to step and typically the flow is ended with a message forwarded to other destinations (queues). The message flows can be characterized as follows:

- Message flows provide the processing sequence required to connect applications.
- Message flows are general-purpose integration applications.
- Message flows are themselves transactional.
- Message flows can be nested and chained together.

The message flow, in contrast to the messaging system that transport the message, must be aware of the message content. The message is parsed in order to determine required actions.

The message broker, such as IBM WebSphere MQ Integrator, supports messages falling into several so-called message domains:

- XML
- JMSMap
- JMSStream
- MRM
- NEONMSG
- Blob

Throughout this section, we use the IBM WebSphere MQ Integrator product as our model message broker.

There are two transaction models supported by the message broker:

- Broker coordinated:
  - The broker controls transaction.
  - When flow is completed, the end of message processing takes place, committing any database changes made in this flow.
- Globally coordinated:
  - The flow is a distributed (global) transaction that is coordinated by the MQ-based transaction manager.
  - The database updates and MQ messages are processed within the same unit of work.

This model is illustrated in Figure 5 on page 26.

The basic unit of processing within a broker is called a *node*. Nodes take input messages, operate on them, and propagate them to other nodes. In Figure 5, the MQInput node receives an initial message. This node is responsible for implementing the transactional model. If the message flow is globally coordinated (the bottom part of the illustration), the node initiates the global (XA transaction). Since this is an MQ-based message broker, the MQBegin call would be made. The MQInput node will ultimately get control again at the end of message flow processing, regardless of any errors or exceptions.

During the message flow processing, many other nodes will get the opportunity to process and propagate messages. There can be nodes of different types, such as message filters, database nodes, compute nodes, and MQOutput nodes.

If the message was processed successfully, then all active database transactions will be committed and the global unit of work will be terminated (committed) using the MQCmit call. In the case of an exception/error, the active database transactions are rolled back and the global unit of work is rolled back with the MQBack call.

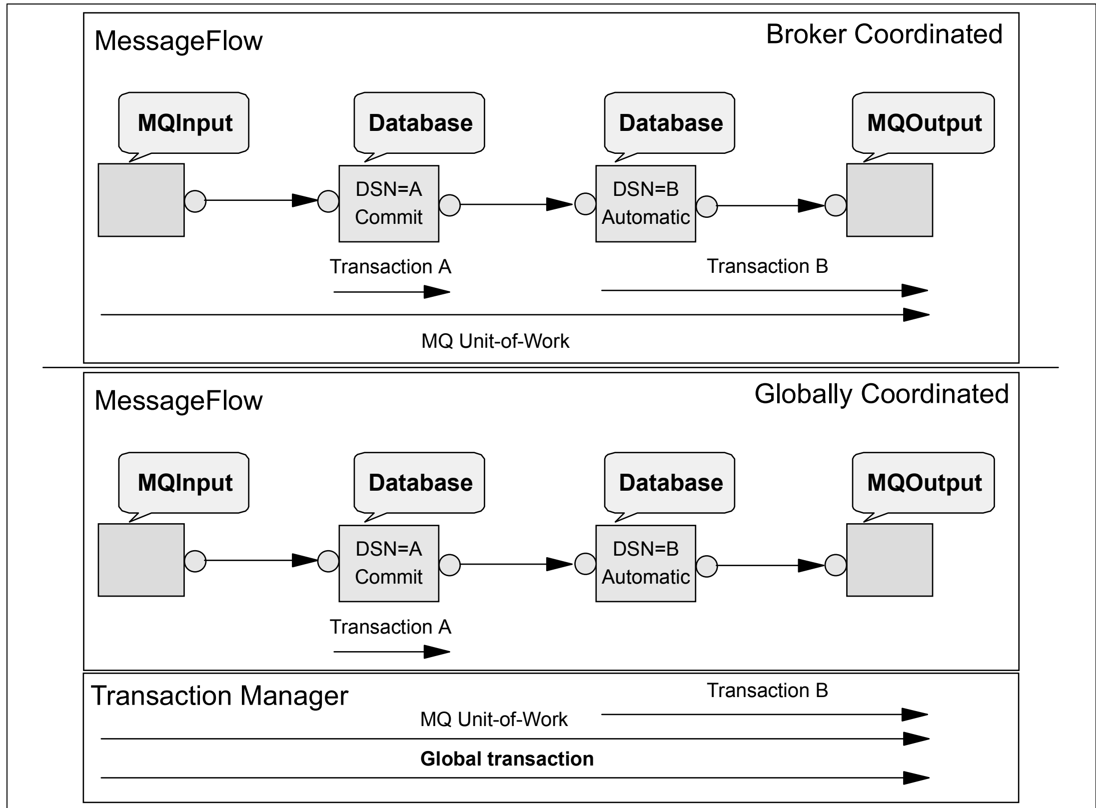


Figure 5 Message Broker WebSphere MQ Integrator transaction model

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

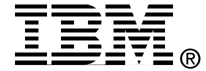
Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright IBM Corp. 2003. All rights reserved.

This document created or updated on June 11, 2004.

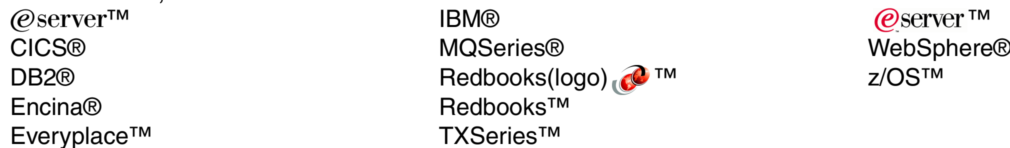


Send us your comments in one of the following ways:

- Use the online **Contact us** review redbook form found at:
  ibm.com/redbooks
- Send your comments in an Internet note to:
  redbook@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HZ8 Building 662
  P.O. Box 12195
  Research Triangle Park, NC 27709-2195 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:



eserver™	IBM®	eserver™
CICS®	MQSeries®	WebSphere®
DB2®	Redbooks(logo) ™	z/OS™
Encina®	Redbooks™	
Everyplace™	TXSeries™	

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.