SAP White Paper
SAP™ NetWeaver

Distributed Transactions and Two-Phase Commit

Armand Wilson

RIG Senior Consultant

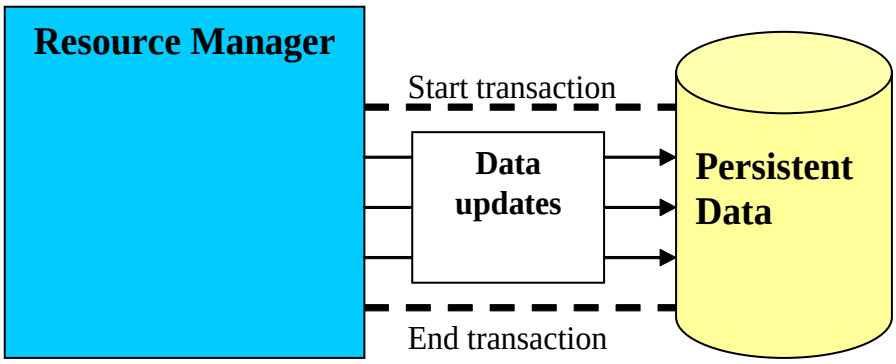
Index and Table of Contents

1. Overview

Transactions are used frequently in database programming. If you are updating multiple entries in a database and they do not all succeed, it is preferable to have all updates fail and for the database to return to its pre-transactional state. How do you make updates to two different databases succeed or fail as a unit? This is the domain of distributed transactions. This paper will describe various strategies for distributed transactions, the advantages and disadvantages of each, and specifically how these strategies can be implemented within the SAP NetWeaver™ framework.

1.1. Transaction Definition

First, let's define transactions. A *transaction* is a set of related tasks that either succeed or fail as a unit. It is important to distance ourselves from thinking that the concept of transactions only pertains to databases. A database is just a specific type of resource manager. A *Resource Manager* is any entity that is responsible for maintaining some resource that has value to one or more parties. Of course in our case, we are interested in software resource managers that manager data about entities of interest. In all cases the data will be persistent. This means the data is stored in a permanent manner, such that the data is available in the event the resource manager software is not up and running due to any software or hardware failures.



For a transaction to succeed (or commit), all participants must guarantee that any change to the data must be persistent. There are four guarantees associated with a transaction called the *ACID properties*. ACID stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

1.1.1. Definition of ACID Properties

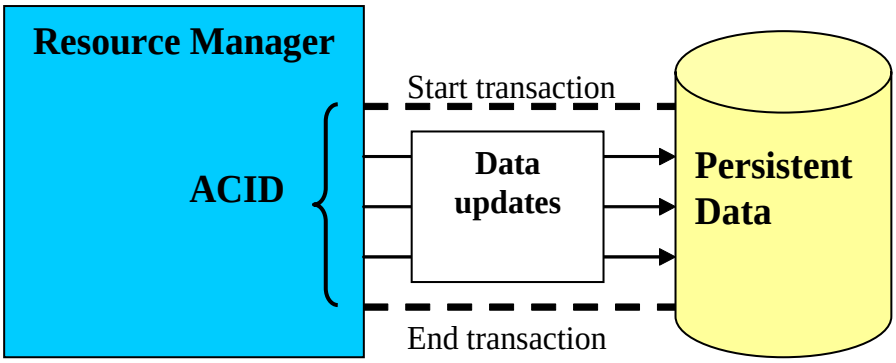
Atomicity guarantees that many operations are bundled together and appear as one contiguous unit of work, operating under an all-or-nothing paradigm—either all of the data updates are executed, or nothing happens if an error occurs at any time. In other words, in the event of failure in any part of the transaction, all data will remain in its former state as if the transaction was never attempted. In transactional terminology, this is referred to as *rolling back* the transaction.

Consistency guarantees that a transaction will leave the system in a consistent state after the transaction is completed. The meaning of consistency varies depending on the logic of the system; it is somewhat up to the application developer to enforce the specific rules governing the consistent state.

Within a transaction, it is possible for some pieces to be in an inconsistent state. However, once the transaction is completed—either successfully or unsuccessfully—the system must return to a consistent state. An example most of us can relate to is a software application installer. Installers write and update files on your hard drive. If you should turn off your computer in the middle of an installation you may be unable to continue the installation or uninstall the program without some manual manipulation of your file system and/or system registry. The installation of the software was left in an *inconsistent* state. Atomicity helps enforce that the system always appear in a consistent state.

Isolation protects concurrently executing transactions from seeing each other's incomplete results. Isolation allows multiple transactions to read or modify data without knowing about each other because each transaction is isolated from the others. This is achieved by using low level synchronization protocols (locking) on the underlying data. There are several levels of isolation available, each with benefits and drawbacks. For example, at the lowest level of isolation, as the data is being changed in the transaction, other users of the data will be exposed to the changes. Thus, if the transaction is rolled back, the other users of the data may see data that will not be accurate a few moments later after the roll back occurs. At higher levels of isolation, other users of the data will not be able to read the data until the transaction is successfully completed or is rolled back.

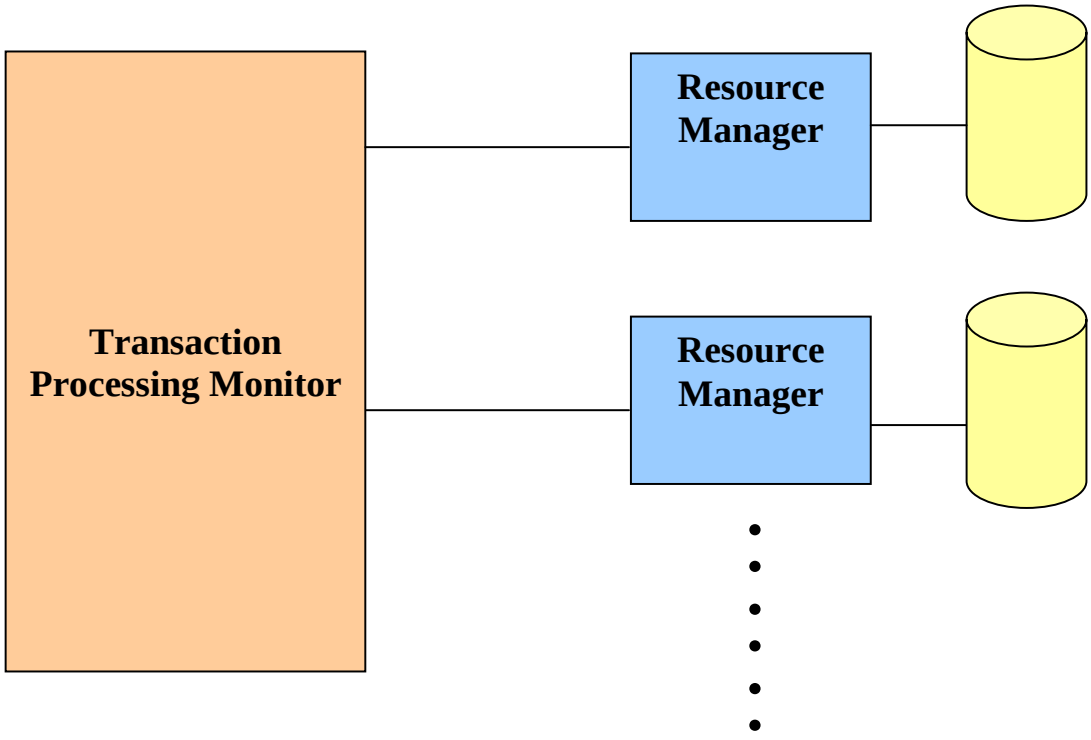
Durability guarantees that updates to managed resources survive failures. Failures include machine crashes, network crashes, hard disk crashes, and power failures. Recoverable resources keep a transactional log so that the permanent data can be reconstructed by reapplying the steps in the log.



2. Distributed Transactions

Distributed transaction processing systems are designed to facilitate transactions that span heterogeneous, transaction-aware resource managers in a distributed environment. The execution of a distributed transaction requires coordination between a global transaction management system and all the local resource managers of all the involved systems. The resource manager and *transaction processing monitor (*or *TPM* as used herein) are the two primary elements of any distributed transactional system.

To support advanced functionalities required in a distributed component-based system, monitor separation from the resource managers is required. The TPM is responsible for managing distributed transactions by coordinating with different resource managers to access data from several different systems. Since multiple application components and resources participate in a transaction, it is necessary for the TPM to establish and maintain the state of the transaction as it occurs. Resource managers inform the TPM of their participation in a transaction by means of a process called *resource enlistment*. The TPM keeps track of all the resources participating in a transaction and uses this information to coordinate transactional work. The TPM has to monitor the execution of the transaction and determine whether to commit or roll back the changes made to ensure atomicity of the transaction.



2.1. Long-lived And Short-lived Transactions

To add to the complexity of distributed transactions, there are categories such as short-lived, long-lived, synchronous, and asynchronous distributed transactions. What is the nature of all this terminology and why is it important?

Synchronous and *asynchronous* refer to the mode of operation of a distributed transaction. If each step of the TPM's communication with the resource managers occurs within the same session context, then the transaction is synchronous. A characteristic of synchronous transactions is *timeout*. Synchronous transactions are intended to complete within a small, finite unit of time on the order of seconds. Thus, if one of the resource managers does not respond to a request from the TPM within a certain timeout period (say several seconds, or a few minutes) the transaction will fail. Asynchronous transactions occur in separate sessions that do not require the success of one to be serially tied to another. Thus, calls can be made to each resource manager in succession without having to wait for the previous one to respond with a return code or to fail in a timeout situation. The success or failure of asynchronous transactions is determined quite differently than for synchronous transactions. Usually some trigger mechanism indicates when all resource managers have responded, or, a polling mechanism checks periodically to see if the resource managers have responded.

Long-lived and short-lived distributed transactions are usually synonymous with asynchronous and synchronous distributed transactions, respectively. The terminology is usually used in regard to the expected response time of the participants in the distributed transaction. Thus, by their nature, long-lived distributed transactions do not lend themselves

well to synchronous use because locking resources for long, sometimes indeterminate time periods, is not acceptable in most applications. In this document, the terms *long-lived* and *asynchronous* distributed transactions are used interchangeably.

2.2. Two-Phase Commit Protocol

For the distributed transaction to be ACID-compliant, all the resource managers spanning the transaction and the TPM must enforce the ACID properties. Traditionally, this is accomplished using the two-phase commit (2-PC) protocol. The 2-PC protocol ensures that all resource managers either all commit to completing the transaction or they all abort the transaction, thus leaving the state of their resources unchanged from the pre-transactional state. When an application requests a transaction to commit, the TPM issues a PREPARE_TO_COMMIT request to all the resource managers. Each of these resources managers must reply to the TPM indicating whether it is ready to commit or not. Only when all the resource managers are ready for a commit does the TPM issue a commit request to all resource managers.

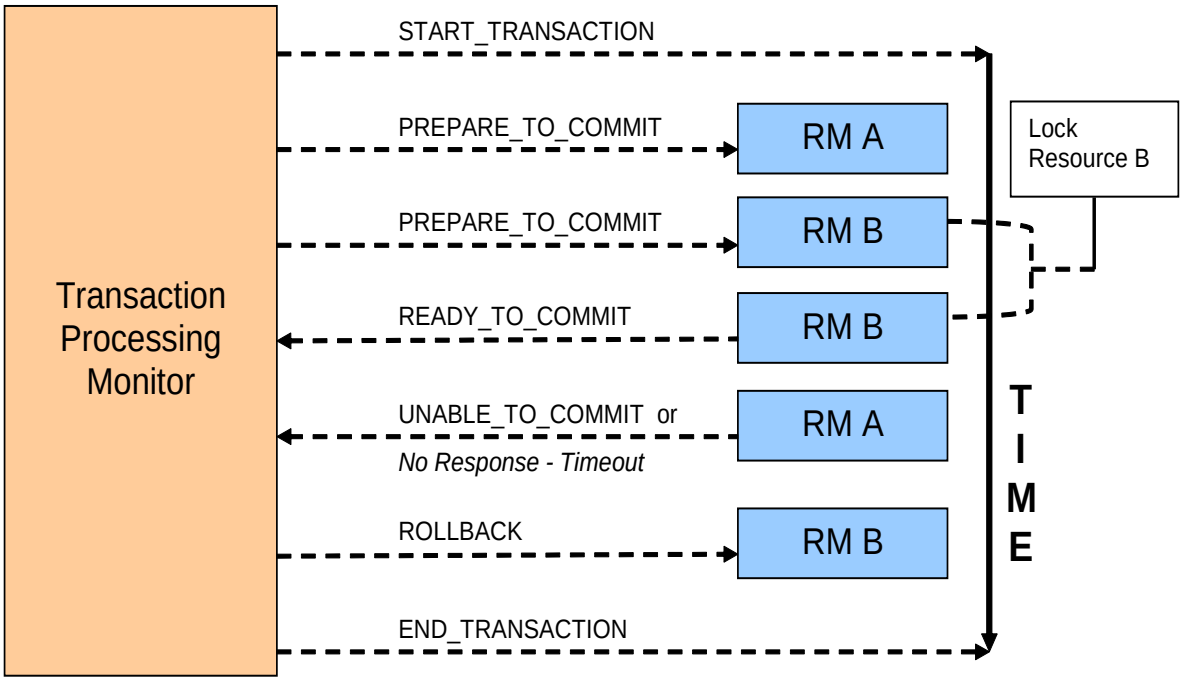
If any resource manager does not respond positively to the PREPARE_TO_COMMIT request, either because the resource is not available or the resource manager is not reachable, the TPM issues a ROLLBACK request to all the applicable resource managers. The distributed transaction is successfully completed and permanent at the point when all resource managers successfully commit their portion of the distributed transaction and acknowledge this to the TPM. The astute reader may note that even after issuing a COMMIT request to each resource manager, there is a remote possibility one might not respond with a successful commit because the resource manager became unavailable sometime between its response to the PREPARE_TO_COMMIT request and the TPM's request to COMMIT. More on this will follow.

It is important to note that the 2-PC protocol is synchronous in nature and thus not well suited for long-lived transactions (see section. 2.1.)[1]

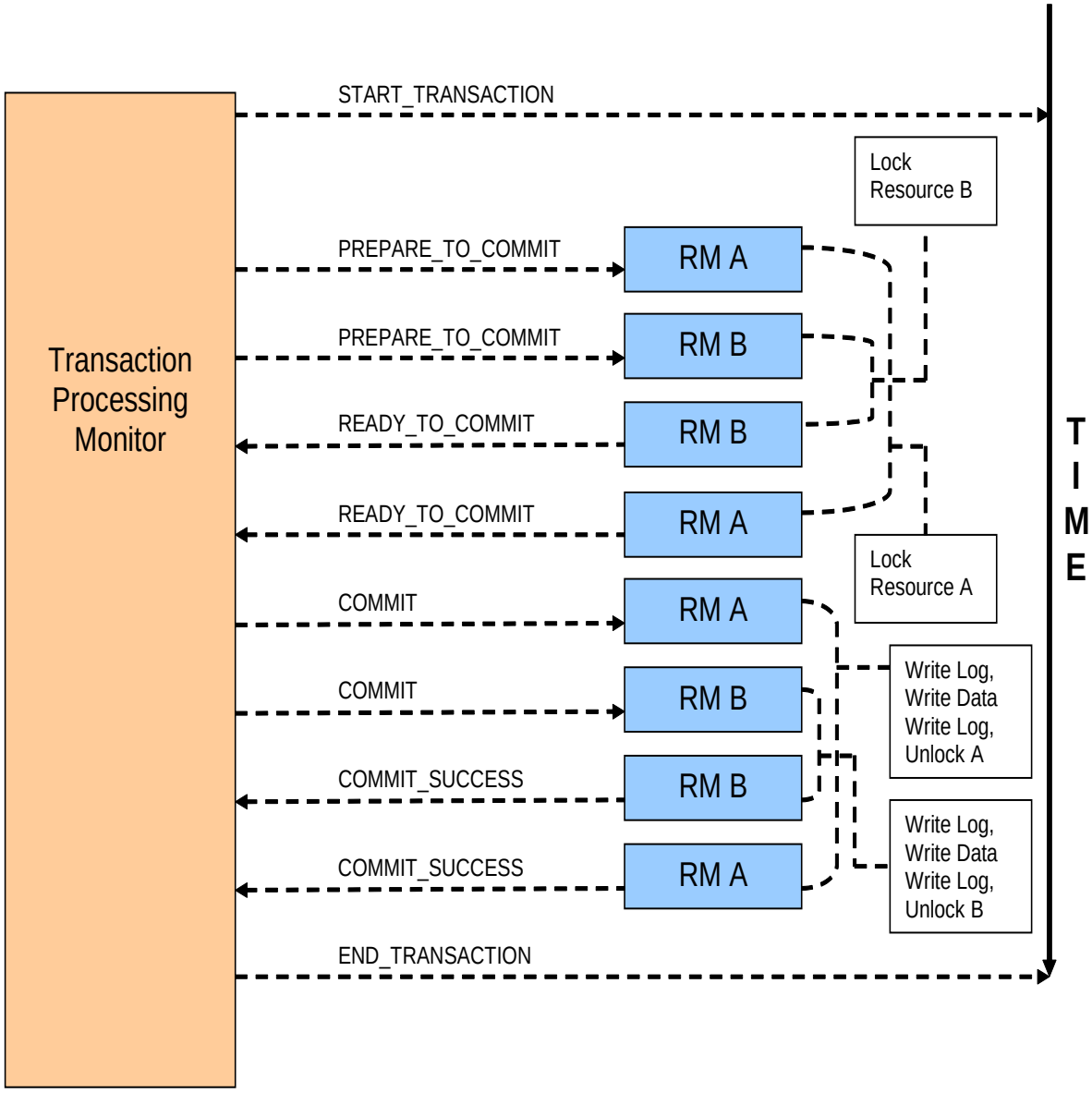
Although 2-PC provides autonomy of a transaction, the required processing load is rather heavy. The transaction speed is always limited by the resource manager with the slowest response, and the network traffic and latency is double that of a normal transaction because of the intermediate, PREPARE_TO_COMMIT request. Although 2-PC provides some reliability in achieving ACID compliance for distributed transactions, as insinuated in the previous paragraph, this is not a guarantee. We shall see later that 2-PC is not "bullet-proof."

[1] To add to the confusion about synchronous and asynchronous behavior, those who examine the 2-PC protocol in detail will notice that asynchronous operations are allowed during a 2-PC transaction. However, these operations must be non-transactional (e.g., you can asynchronously write to a message queue, etc., as long as no resource managers are involved that are included in the distributed transaction.)

The diagram below shows a failed 2-PC transaction resulting from resource manager (RM) "A" being unable to commit its resources for the transaction or having not responded within the given timeout period to the PREPARE_TO_COMMIT request. Note that after receiving the PREPARE_TO_COMMIT request and prior to responding back to the TPM, RM "B" placed a lock on all the required resources in its portion of the distributed transaction. These resources were subsequently released (unlocked) after RM B received the rollback request from the TPM.



The diagram below shows a successful 2-PC transaction.



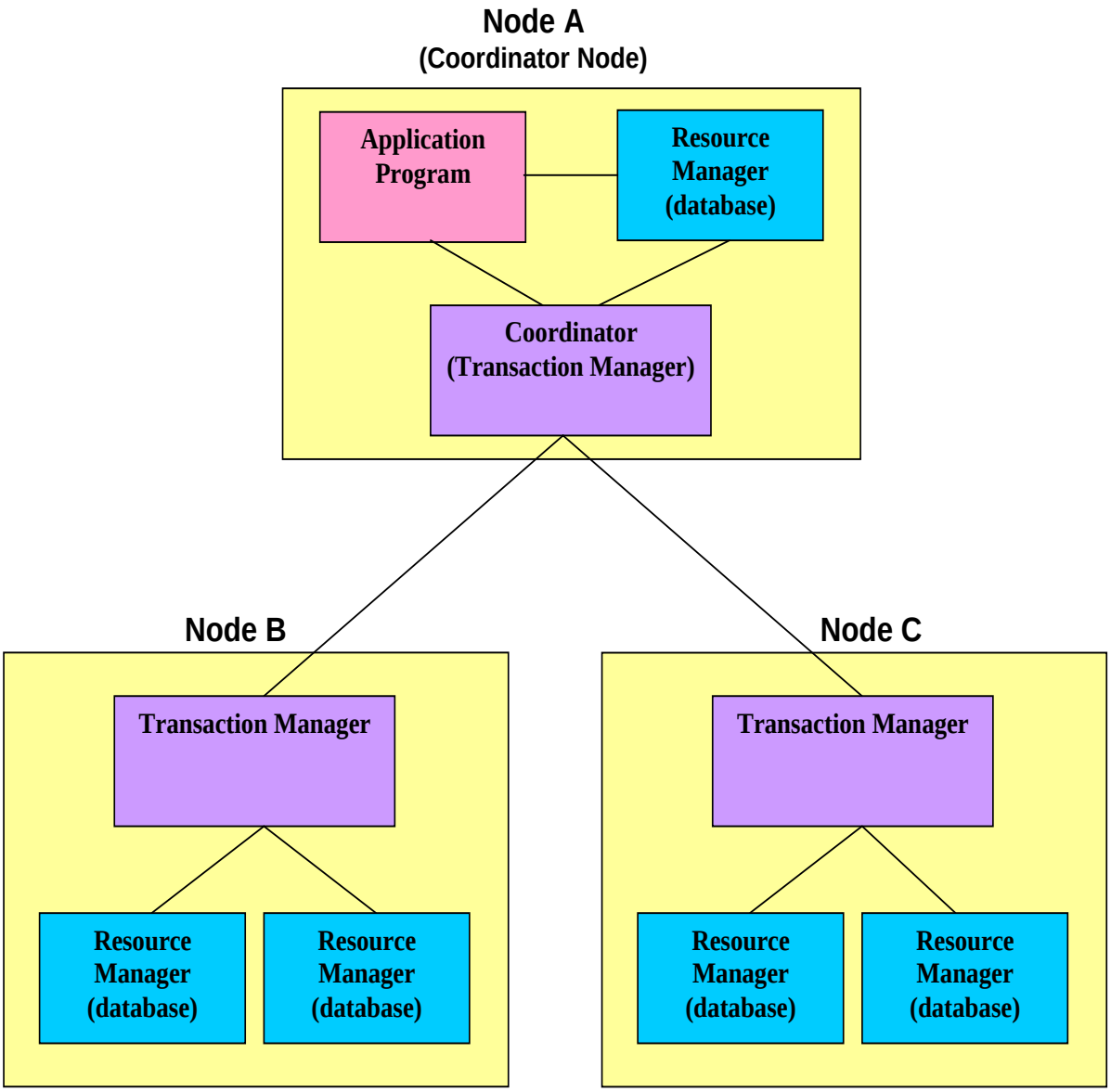
2.2.1. Elements of a Two-Phase Commit System

Definitions for the various elements of a 2-PC system are provided below.

- **Application**
  Software can be defined as a program or group of programs designed for end users. Software can be divided into two general classes: *systems software* and *applications software*. Systems software consists of low-level programs that interact with the computer at a very basic level. This includes operating systems, compilers, and utilities for managing computer resources. In contrast, application software (also called *end-user programs*) includes database programs, word processors, and spreadsheets. Figuratively speaking, application software sits on top of systems software because it is unable to run without the operating system and system utilities.
- **Resource Manager (RM)**
  The resource manager is usually a database management system, such as Oracle, DB2, or SQL Server. A resource manager is responsible for maintaining and recovering its own resources. From the perspective of the application, the resource manager is a single attachment to the resource (e.g., a database). Note that resource managers are not limited to databases. Any software program that manages persistent data is a resource manager.
- **Transaction Manager (TM)**
  The transaction manager coordinates the actions of the resource managers that are located on the same node (local resource managers) as the transaction manager. (A transaction manager may also act as the coordinator under specific circumstances.)
- **Transaction Coordinator (TC)**
  The transaction coordinator is the transaction manager on the node where the application started the transaction. The coordinator orchestrates the distributed transaction by communicating with transaction managers on other nodes (remote transaction managers) and with resource managers on the same node (local resource managers).
- **Transaction Processing Monitor (TPM)**
  The transaction processing monitor consists of the transaction coordinator and all the transaction managers composing the distributed 2-PC system.

2.2.2. Landscape of a Two-Phase Commit System

The landscape of a 2-PC system is shown in the diagram below.



2.2.3. The Two-Phase Commit Process

When the application starts a distributed transaction, the TM on the same node becomes the TC. Following are the steps that involved in consummating the distributed transaction.

1. The TC first checks that the TM software is running on all the nodes participating in the transaction. If the TM software is not running, the TC returns an error and does not start the distributed transaction.
2. If all the TM's are available, the TC generates a distributed transaction identifier and associates the identifier with all the participants in that particular transaction.
3. When the application is ready to commit all the changes to the RMs involved in the distributed transaction, all the nodes in the transaction must execute both phases of the two-phase commit protocol, the *prepare* phase and the *commit* phase.
4. During the prepare phase, the TC asks each RM participating in the transaction whether or not it is prepared to commit the transaction. If the TC receives a "yes" response from *all* the RMs, the TC instructs the participants in the transaction to enter the commit phase.
5. During the commit phase, the TC instructs the RM to make permanent changes to its data, i.e. to commit the changes. The RM then commits the changes and the transaction is completed.

The table below describes the circumstances under which distributed transactions are committed or rolled back under the 2-PC protocol.

When This Happens:	**This is the Result:**
Application instructs the transaction to roll back.	Transaction rolls back
Process or image failure occurs before all participants vote.	Transaction rolls back
Any participant votes no.	Transaction rolls back
All participants vote yes and no image failures.	Transaction commits
Process or image failure occurs after all participants have voted and the TC has received all yes votes.	Transaction commits but is unresolved

2.2.4. A Standard Protocol for Two-Phase Commit

It is not enough to list the steps in the 2-PC process. A protocol must be established to ensure transparency between RMs, TM's, and TCs in a distributed transaction. This means that any transaction monitor or transaction coordinator of a TPM can engage in a distributed transaction with any resource manager. This is referred to as interoperability between elements of any 2-PC system. Currently, the most widely used open standard is the X/Open Distributed Transaction Processing (DTP) model. X/Open DTP was proposed by the Object Management Group, and is a standard among most of the commercial vendors providing transaction processing and relational database solutions. It is an optional CORBA service.

The two major interfaces specified in the DTP model are the TX and XA interfaces. The TX interface is between the application and the TPM and is implemented within the TPM. It provides transaction demarcation services by allowing the application components to bind transactional operations within global transactions. The XA interface defines the interface between RMs and TMs. When both the TM and RM ‘s support the XA interface, they can be plugged together and transaction coordination can take place between them. This is the most important interface in the standard and has wide industry acceptance. Commercial transaction management products like TXSeries/Encina, Tuxedo, and TopEnd support the TX interface. Most of the commercial databases such as Oracle, Sybase, Informix, and MS SQL Server, and messaging middleware like IBM’s MQSeries, also support the TX interface.

In the past, SAP systems have always operated within the context of a single database and therefore without the necessity of 2-PC. The SAP® Web Application Server (SAP WAS) 6.20 and later versions support the XA interface and 2-PC in the Java runtime. However, the MaxDB (SAP DB in pre-6.40 releases of SAP WAS) does not have the capability to participate in 2-PC transactions. This leaves a false impression of the SAP WAS 2-PC capability. In reality, the SAP WAS has the same capability to participate in 2-PC as all other high-end commercial application servers.

The table below lists the basic interface for the 2-PC XA protocol:

`xa_open`	Connects to the resource manager.
`xa_close`	Disconnects from the resource manager.
`xa_start`	Starts a new transaction and associates it with the given transaction ID (XID), or associates the process with an existing transaction.
`xa_end`	Disassociates the process from the given XID.
`xa_rollback`	Rolls back the transaction associated with the given XID.
`xa_prepare`	Prepares the transaction associated with the given XID. This is the first phase of the two-phase commit protocol.
`xa_commit`	Commits the transaction associated with the given XID. This is the second phase of the two-phase commit protocol.
`xa_recover`	Retrieves a list of prepared, heuristically committed or rolled back transaction.
`xa_forget`	Forgets the heuristic transaction associated with the given XID.

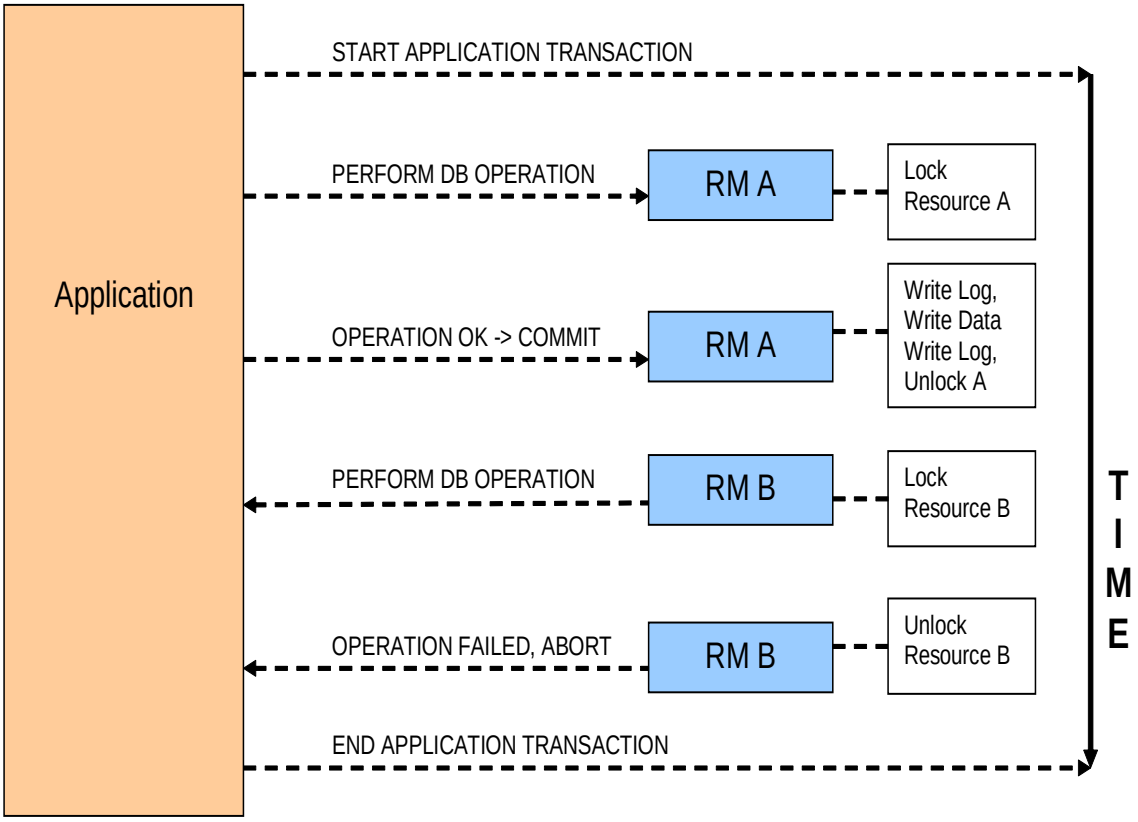
It is important to note that a widely adapted standard, such as the DTP model, is of critical importance to the success of 2-PC. Without this interoperability, custom programming would have to be implemented for each RM enlisted in 2-PC transactions. Of equal importance: Only when a RM becomes a commodity is its acceptance of a 2-PC standard of any practical consequence.

This may seem convoluted, but it is very tied to the reality of the software marketplace. When databases were new technology, few applications were using these and the consequence of any one database implementing a standard such as 2-PC was not really important. It would be unusual for an application to use one database, much less two databases. When databases became a commodity, only a limited number survived in the marketplace and all of these had to implement the latest and greatest standards to survive. Only then does the practical application of technology, such as 2-PC, become important.

That is why it is important to make this observation: *At the time most legacy software in use today was developed, the only commoditized resource managers were databases. Thus, for these legacy applications, 2-PC is only important for conducting distributed transactions at the database level of abstraction.*

2.2.5. Distributed Transactions Without 2-PC

To understand the value of 2-PC in distributed transactions, let us examine a process that uses two resource managers without the 2-PC protocol. This is referred to here as serialized database access. The process simply performs a transaction on each RM in sequence and either commits or aborts the transaction based on the return code from the RM.



From the diagram above you can deduce the primary advantage of 2-PC. Employing serialized database access offers no protection against primary transactional failure. This refers to the case where the resource manager is not reachable or fails before the requested resource is locked. When RM B fails to perform its transaction, the overall application fails but there is no way for the application to roll back the transaction committed on RM A. Therefore, the resources managed by RM A and B have become inconsistent. If RM A was a transaction to debit an account, then the money that was to be credited to the account managed by RM B was lost. The only recourse is for the transaction committed by RM A to be manually undone. In this case an account manager with "super-user" privileges will have to manually credit the account of RM A to perform a manual rollback operation, or credit the account of RM B to complete the overall application operation. Written policies will dictate the conditions and actions to be taken by the super-user account manager to recover failed transactions. Policies for manual recover of failed transactions are not only necessary for non-2-PC distributed transactions. We shall see in the next section that recovery policies are also necessary for 2-PC systems as well.

2.2.6. Possible Failure: The Unresolved Transaction

The intermediate voting phase in the 2-PC process leaves a lesser likelihood of transaction failure than distributed systems without 2-PC. However, 2-PC is not 100 percent reliable because there is a remote possibility that one or more RMs fail between the vote and commit phase. For example, the network connection may be lost or the node itself may suffer some irreparable damage. If these situations occur after the participants vote but before transactions are completed on all nodes, the distributed transaction is referred to as *unresolved*. If you cannot wait for the connection to the TC to be restored or if the damage cannot be repaired, you can force the distributed transaction to commit or roll back. The recover process is manual and varies from TPM to TPM. One scenario is presented below for illustration (Note: Don't try to understand the details; the purpose here is to illustrate the complexity of fixing an unresolved 2-PC transaction.)

1. Use the State=Blocked qualifier of the RMU Dump Users command. This command generates a list of unresolved transactions for a particular database. Depending upon your application, you might need to generate the list for more than one database. The following example generates a list of all unresolved transactions for the personnel database:

```
$ RMU/DUMP/USERS /STATE=BLOCKED  personnel

        >Blocked user with process ID 000000C5
        >Stream ID is 1
        >Monitor ID is 1
        >Transaction ID is 1
         >Recovery journal filename is
         >"DISK1:[USERA]PERSONNEL$0094FF5D19B7E2A0.RUJ;1"
        >Transaction sequence number is 104
        >DECdtm TID is  0019F490 0019F494 0019F498 0019F49C
        >Failure occurred on node DBTWO
        >Parent node is DBONE
```

2. Examine the list to identify transactions that need to be resolved. (Remember that a database can be involved in more than one unresolved transaction.) Note the transaction sequence number associated with these transactions. In the example shown in Step 1, the personnel database is in a blocked state; that is, changes have neither committed nor rolled back and the database is involved in an unresolved distributed transaction. The transaction sequence number is 104. If the database is involved in more than one unresolved transaction, the command would generate output for each unresolved transaction.
3. Refer to your application to determine which databases are affected by the unresolved transactions.
4. Consult with the database administrators for all of the databases affected by the unresolved transactions to determine how to resolve them. Collaboration is necessary because the transaction might have completed on one node before the coordinator became unreachable to the other nodes. When this occurs, the transaction must be altered to the same state on all nodes affected by the transaction.

5. Use the RMU Resolve command on the database to complete the unresolved transactions. For example, to complete the unresolved transactions for the personnel database, and confirm and log your action, enter the following command:

```
$ RMU/RESOLVE/LOG/CONFIRM  personnel

        >Blocked user with process ID 000000C5
        >Stream ID is 1
        >Monitor ID is 1
        >Transaction ID is 1
         >Recovery journal filename is
         >"DISK1:[USERA]PERSONNEL$0094FF5D19B7E2A0.RUJ;1"
        >Transaction sequence number is 104
        >DECdtm TID is  0019F028 0019F02C 0019F030 0019F034
        >Failure occurred on node DBTWO
        >Parent node is DBONE

Do you wish to COMMIT/ABORT/IGNORE this transaction: COMMIT
Do you really want to COMMIT this transaction? [N]: Y
%RMU-I-LOGRESOLVE, blocked transaction with TSN 104 committed
```

If the database is involved in more than one unresolved transaction, Oracle RMU asks what action you want to take for each transaction. If you want to commit or abort *all* unresolved transactions for the database, you can specify the action in the command line by using the State qualifier. For example, to commit all the unresolved transactions involving the personnel database, use the following command:

```
$ RMU/RESOLVE/CONFIRM/LOG /STATE=COMMIT  personnel
```

6. Check to see that all the transactions are resolved:

```
$ RMU/DUMP/USERS/STATE=BLOCKED personnel
        >No blocked users
```

2.2.7. Do Not Confuse Rollback with 2-PC

It is not uncommon to confuse the concepts of rollback and 2-PC. Rollback is usually present in transactions and provides a means of undoing previous work prior to committing changes to become permanent (persistent). All database transactions provide a means of rollback via a log record of the changes made to the database during the course of the transaction. For example, several database write operations may be submitted to the database as shown below:

```
BEGIN TRANSACTION
UPDATE  authors
```

```
   SET  au_fname = 'John'
   WHERE au_id = '172-32-1176'
UPDATE  authors
   SET  au_fname = 'Marg'
   WHERE au_id = '213-46-8915'
COMMIT TRANSACTION
```

In this example, two UPDATE operations on the database are written to permanent storage after the COMMIT TRANSACTION command is received by the database. Prior to the COMMIT TRANSACTION, each operation received by the database is only stored in a memory buffer and written to a log file on the hard drive. An important implicit action is also taken by the database on behalf of the client. For review, read the description of ACID properties in section 1.2, in particular, the definition of *isolation*. Isolation is implicitly enforced by the database based on the level of isolation set by default or explicitly set by the client. This determines what database records are locked for reading and/or writing by other clients that are concurrently trying to access the same data. All database records locked during the course of the transaction are also recorded in the log file. When the COMMIT_TRASACTION operation is received, the UPDATE operations (stored in a memory buffer of the server) are actually written to the database records on the file system. Each successful operation is also written to the database log file. In this way, if the database server fails at any time (even during the commit phase) the transaction can be returned to a fully consistent state when the database recovers by resolving the transactional process recorded in the log file.

If an intermediate result of a transaction fails, the user may wish to rollback the transaction, as shown below:

```
BEGIN TRANSACTION

UPDATE    authors
   SET    au_fname = 'John'
   WHERE  au_id = '172-32-1176'

UPDATE    authors
   SET    au_fname = 'JohnY'
   WHERE  city = 'Lawrence'

IF @@ROWCOUNT = 5
  COMMIT TRANSACTION
ELSE
  ROLLBACK TRANSACTIOn
```

Suppose that for whatever reason, the second UPDATE statement should modify exactly five rows. If @@ROWCOUNT, which is a global counter maintained by the SQL command line tool, is five, then the transaction commits. Otherwise it is rolled back. The ROLLBACK TRANSACTION statement "undoes" all the log entries since the matching BEGIN TRANSACTION statement and removes the operations from the in-memory transaction buffer on the server. In this way, none of the data on the file system is changed and all the implicit locks on the data are removed. As in the case of a COMMIT_TRANSACTION, the ROLLBACK_TRANSACTION operation also writes

to the log file because it is releasing database locks that might need to be resolved should the database server fail during the rollback process.

In the same manner as individual database operations can be grouped together into a compound transaction, so can business operations. Business software exposes higher level APIs for performing business operations. Those APIs that result in changes to a database will be performing an implicit database transaction on behalf of the client.

Most business software, including SAP BAPIs after version 4.0A, provide a means of grouping the results of calling several APIs and either committing or "rolling back" these results. The client indicates the demarcation points that start and commit the transaction. Various techniques are used by the business software vendors to update the database or "rollback" the changes. It is important to note that grouping several calls to the same vendor's API is not necessarily an example of 2-PC. More than likely, a single resource manager is being used to manage the persistent data on the backend. Even in cases where multiple databases may be involved, this capability is only possible within that particular vendor's API, and *no* others. As noted in section 2.1.4, *it makes no sense to discuss the support of 2-PC at the business-level of abstraction because no benefit can be achieved from interoperability.*

So in conclusion, do not confuse *rollback* with implementation of the two-phase commit protocol. Two-phase commit does utilize the concept of rollback in the *prepare-to-commit* phase of the process. However, rollback is an essential part of the transactional process and is not particular to 2-PC.

2.2.8. The Truth About Two-Phase Commit

To better understand the role of 2-PC in modern system integration scenarios, we will first examine some of the motivations for its use in legacy systems. Given the synchronous nature of 2-PC, it has been used in OLTP-type applications such as airline reservations and banking systems. In many cases, the application of 2-PC in legacy systems was *not* driven by system integration. Rather, the system design either intentionally, or as a result of system enhancement, incorporated more than one database. Often, when these legacy systems were built as long ago as the early 1980's, disk drive hardware was extremely expensive, suffered from relatively poor throughput, and thus had none of the features that make databases extremely fast, scalable, and reliable as modern commercial databases. If a single database were used in these business landscapes (as was the case for SAP systems[2]) there would have been no need to implement 2-PC transactions.

The 2-PC protocol is a proven and viable solution for handling database-level distributed transactions. However, is it really a viable solution for business process integration and in the design of new software applications?

[2] SAP was the first large commercial software package to recognize the importance of using a single database and data transparency in its system architecture.

Manual Recovery Policies Are Still Necessary and Complex

First, as illustrated in section 2.1.6, 2-PC does not eliminate the need for manual recovery methods. In fact, the recovery methods may be much more complex than those for non-2PC transactions (e.g., simple serialized access to resource managers). In the case of a failed 2-PC transaction, a database administrator is required to mechanically fix the transaction, while an application that abstracts access to the database as business operations can employ a domain expert (e.g., a bank account manager) to apply manual business methods using previously established policies.

Cannot Use Higher Level of Abstraction with Legacy Systems

Two-Phase commit is a protocol that can be applied to any resource manager that supports it. However, in legacy systems 2-PC is implemented only at the database level of abstraction based on the X/Open DTP model. Therefore, to take advantage of the protocol, we must bypass higher levels of abstraction and deal directly with database calls, thus exposing the database schema.

As noted in section 2.1.4, SAP architecture has always maintained use of a single database in the system landscape. This is a sound approach that promotes simplicity and robustness. Also, this has allowed SAP to unfailingly maintain database schema transparency, also a sound approach to maintaining robustness and for reducing system maintenance and complexity. Exposing database schema to clients has been recognized as a poor approach to system design for many decades, but has been consistently violated by many system designers. In some respects, the need for 2-PC has evolved from poor practices in system design.

System software vendors such as Oracle have been very vocal about their capability to implement 2-PC and are quick to imply a large advantage over SAP. Once we have a good understanding of 2-PC and how it is really implemented in practice, it is easy to recognize that Oracle and other vendors really have no advantage over SAP in this regard. No vendor can offer 2-PC, cross-vendor support of legacy software except at the database level of abstraction (see section 2.2.4). Therefore, it is only possible for Oracle and other vendors to provide 2-PC in the same manner as everyone else… via direct database calls. This is a bad practice that was recognized as such many years ago by SAP and avoided whenever possible. Vendors that advertise their advantages in 2-PC capability at the database level are in essence fueling bad practices that will compromise the robustness of all the systems incorporated in such a scheme.

Mostly Not Suitable for Business Process Integration

Driven by a worldwide economy, mergers and acquisitions, companies are becoming more reliant on system integration. Interest in 2-PC has grown as well because it is often incorrectly viewed as a tried and proven solution for process integration. However, because of its synchronous nature, the 2-PC protocol is frequently not a suitable solution for integrating business processes. Unlike OLTP, integrated business processes are frequently long-lived in nature and resources cannot be locked for the duration of a distributed transaction. A classic implementation of a successful, high volume, distributed business transaction is a trip booking. Say the trip involves an airline seat, a rental car, and a hotel room. For the trip booking to be successful, all three resources must be booked for the specified date. If the airline and rental car are booked successfully but the hotel booking fails, we want the whole transaction to fail (to be rolled back). If this scenario involved a non-refundable airline ticket, another problem will have to be undone by the travel agent. The solution, of course, is the traditional reservation system used by all service industries. Instead of directly booking the trip, we reserve each segment and then book the resources in a "second phase" if all the criteria for the trip are achieved. This is a good example of how alternatives to 2-PC can be achieved using business-level concepts and mechanisms instead of relying on the synchronous nature of the 2-PC protocol.

Of course systems such as these can, and have been, automated with software. An argument can be made that the 2-PC protocol can be applied to this problem as well. Imagine the "reservation" as being the "prepare-to-commit" part of 2-PC. Although this is certainly true, we must ask ourselves it this approach will solve the business problem in the best possible manner. If clients are using the system via a web browser, we must consider cases where the user is interrupted part way through the transaction and the session times-out. Do we really want to rollback all the work and make the client redo everything upon returning? It would be better to save the session information before the session is timed-out and let the user resume where he or she left off before. The reservation approach is a flexible business solution that was invented long before software was used to automate the process. The "reservation," although something of a resource "lock," is much more flexible than the conventional lock of synchronous transactions. The reservation alerts other potential buyers and sellers that a resource has the potential to become available in the future. It is not uncommon for someone to give up a reservation for many reasons. In effect, it is a very "soft" lock. A reservation can timeout—if not confirmed within some timeframe, the reservation is removed. There is no mechanism in the 2-PC protocol to allow for a lock to timeout. (The only timeout applicable to the 2-PC protocol is associated with the response time of the resource manager.)

In conclusion, unless we are dealing with OLTP-type systems in an integration scenario, 2-PC is usually not a viable solution. Furthermore, unless these systems are based on modern resource managers that support 2-PC at the functional level of abstraction (currently only some messaging systems and J2EE meet this criteria), we are forced to deal directly with low-level database calls. Thus, 2-PC will frequently *not* be the optimal solution to our business integration scenarios and we should carefully evaluate other alternatives.

2.3. Modern Design of Distributed Systems

Modern distributed systems have the following attributes:

- **Transparency**: It is not necessary for the user of a system to know where the data is located (database name, location, etc.). Through the use of directory services, it is also not necessary for the user to know the location of the machines that run the software. Most importantly, the user should not have to know how the data such as database names, table names, and database schema is persisted.
- **System Scalability and Reliability**: Severs are replicated in modern systems, so there is no single point of server failure in the system. Traffic to an unavailable server is redirected automatically and can be made transparent to the user. Scalability is achieved because additional servers can be added by simple reconfiguration and without shutting down or interfering with the current operation of the system.
- **Database Scalability and Reliability**: Modern commercial database servers have evolved to become highly scalable and reliable commodities. Through the use of disk stripping, hardware clustering, table fragmentation, and other techniques, databases can make effective use of hardware. Disk storage is very inexpensive, highly reliable, and throughput rates are outstanding. All of these factors are continuing to improve despite all the gains from the past.
- **Failover**: Although replication is an essential part of modern distributed systems, 100 percent replication has been recognized as a performance bottleneck. Attempts to replicate database servers along with application servers in multi-tier systems results in huge network traffic and latency. This limits scalability with only marginal or possibly negative improvement in reliability. Services requiring global synchronization such as lock servers and database servers are typically single points of failure in modern systems. However, with modern hardware clustering techniques, this is no longer an issue. Since this clustering is done via hardware services, there is little loss in performance.
- **Replication**: For systems where users are distributed over a large geographic area and where system throughput is high, it may be possible and desirable to replicate data over several distributed databases. To be efficient, the data involved must be able to tolerate some “staleness” (the data can be a little out of date without dire consequence to the system operation). For example, a system that shares medical data about patients can be slightly out of date (say, by hours). In this case database replication can be done once or twice a day, usually during light system load. On the other hand, an inventory management system might not be a good candidate for system replication. If an item sold at one site of a distributed database system results in zero stock, it may be necessary for the all the distributed databases to have this information. Otherwise, items may be sold that do not exist in inventory. Of course,

this situation can sometimes be handled by a messaging system that informs purchasers of backorders and gives them the opportunity to cancel the order if desired.

2.3.1. 2-PC in J2EE

CORBA and J2EE are the first standards to incorporate 2-PC protocol at a functional level of abstraction. J2EE provides a standard interface for a transaction manager and a standard infrastructure to support EJB containers and Web containers as resource managers within the J2EE architecture. J2EE provides a declarative and programmical interface based on the X/Open, XA standard. The interface provides for transaction demarcation at the functional level of abstraction. Enterprise Java Beans (EJBs) can use J2EE's Java Transaction Service (JTA) to encompass their function calls (methods) within transactional boundaries. Of course, the underlying resource managers for the data touched by these methods (e.g., databases) must also support the XA 2-PC protocol. Any *Java* applications running on the SAP J2EE Engine can implement 2-PC at the functional level. This is on-par with WebSphere, WebLogic, and all other J2EE compliant applications servers.

Besides managing the transactional behavior of EJB containers and Web containers, JTS can serve as a transaction manager for the Java Messaging Service (JMS) and the Java Connector Architecture (JCA.) In addition to the JTS, there is also a component called the Java Transaction API (JTA). The JTA is mostly used for programming by J2EE server vendors but also has some API interfaces for user by application developers. For simplicity, the details of JTS and JTA will be left out of this discussion. See Appendix - Java Transaction Management for additional details.

EJBs, Java messaging, and Java connectors are often used as *wrappers* over legacy code so the modern features of J2EE can be leveraged. Also, if transactions are involved, these wrappers can take advantage of the logical resource management features of J2EE rather than dealing directly with the database level of abstraction. However, it is important to realize that the same principles apply to 2-PC under the framework of J2EE as with the original legacy systems that are under the wrappers. It is not possible for J2EE and JTS to raise the level of abstraction for existing legacy code that is eventually dependent upon 2-PC for transactional management. The underlying legacy code will still be just as brittle and inflexible as before and will continue to be a bottleneck to expanding the features of the legacy system.

Java and J2EE enable much easier incorporation of 2-PC and allow us to program at a level of abstraction suitable for modern system design. However, the tradeoff of using 2-PC versus other techniques should be carefully evaluated as noted in section 2.2.8. In most cases, even when dealing with present-day system software and hardware, new applications should avoid the higher overhead associated with 2-PC. In some cases, due to architectural constraints, it may be sensible to use 2-PC, such as the case of a JMS application that writes to a database. However, even for this scenario, if the architect has the flexibility to use Message Driven Beans, he or she could leverage the asynchronous behavior of messaging and avoid 2-PC. In most cases, this would result in higher throughput and system reliability.

In closing this section, note that JTS is the general purpose transaction manager for the J2EE framework and handles both distributed and local transactional behavior. For those interesting in how the object-oriented nature of J2EE is employed to manage distributed transactions, see the Appendix - Java Transaction Management at the end of this document.

2.3.2. Distributed Transactions for Modern Systems

Distributed business processes today are handled by a variety of techniques including Web services, JMS, JCA, and proprietary techniques such as SAP JCO. Each has their own advantages and disadvantages. JMS can be configured to be synchronous or asynchronous and to have guaranteed delivery of messages. JCA generalizes the concept of resource adapters in J2EE to deal with any EIS system, TPM, and other legacy systems. The advantages of Web services, based on HTTP protocol and XML messaging, are contributing to its rapid acceptance as the standard for distributed business transactions.

Eventually, every distributed process has to deal with failures and determine policies to deal with them. The types and sophistication of policies are often a function of the significance of these failures relative to the overall success of the distributed process. Most policies are manual in nature and are triggered via some alert mechanism system such as email. The preferred scenario is to have these failures repaired by someone who is an expert in the business process. Since hardware and software advances through the years have greatly improved the reliability of business systems, manual recovery techniques are usually adequate.

In cases where low system reliability results in high cost associated with manual recovery techniques, and/or seriously erodes the confidence of users and clients, it is usually best to investigate and fix the causes of the reliability problems rather than invest in sophisticated mechanisms to automate failure recovery. Also, every manager in the decision process to automate failure recovery of distributed systems must recognize the limitations of complex software. No matter how sophisticated auto-recovery techniques become, there is always a need for manual recovery policies. Since these policies and the infrastructure to execute these are necessary in all cases, the trade-off becomes one based on the following factors.

Cost-Risk Factors in Choosing Recovery Mechanisms

- The cost and risk of increasing system reliability by conventional means, such as rewriting troublesome software, re-architecting systems to use modern multi-tier architecture, updating key system software such as databases, and updating hardware and OS's to use clustering techniques.
- The cost and risk of training and maintaining domain experts in the business process to manually repair software failures, including the costs of alert and reporting systems to assist in this effort.
- The cost and risk to design and implement automated mechanisms for failure recovery.

2.3.3. Web Services: Hope for Business Integration

Web services are fairly new in the mix of techniques for system integration. However, Web services have some benefits that make them very attractive for long-lived, non-OLPT type integration applications. In the table below, the risk and concerns of process integration are listed in the first column and the benefits drawn from Web services in the second column.

Parties need to communicate with each other using different information systems.	XML (Extensible Markup Language) makes data portable. Tools to process and manipulate XML are ubiquitous across programming languages and operating systems.
Parties need to communicate a protocol that is platform-independent and extensible.	SOAP (Simple Object Access Protocol) XML-based, extensible protocol. Uses ubiquitous HTTP used as presentation protocol.
Clients must be able to discover and locate services and services must be easily invoked programmatically by clients.	WSDL (Web Service Description Language) is a special form of XML that contains all the information a client needs to programmatically invoke a Web service. UDDI (Universal Description, Discovery and Integration) business registries can be used to index WSDL documents so these are searchable
Communication must be secure and trusted.	WS-Security (Web Service Security) is a standard that envelops industry standards such as X.509, Kerberos and SSL to provide secure communications.
Technology must be scalable and highly available.	Built on Internet Infrastructure.

Despite apparent benefits, Web services for business integration introduce a number of risks just by exposing internal systems to access by others. When business processes are automated by Web services, accidental or even intentional abuse can easily go

undetected. For example, imagine how an unethical travel aggregator might exploit an airline reservation Web service. Months in advance, the aggregator reserves every available seat on a particular flight—but at the last minute, cancels them. In a panic to sell the seats, the airline puts them on sale at a deep discount. The unscrupulous travel aggregator then repurchases the same seats at this much lower price.

Accepting a reservation carries an inherent risk of such a last-minute cancellation. This problem exists even without Web services, but there are systems in place to detect and prevent most abuses. Airlines manage this risk by overbooking. Concert and theater ticket agencies protect themselves using no-refund policies. But many other businesses—particularly those in wholesale trade—have no formal methods for managing cancellation risks. The risks and abuses of cancellations will probably increase and spread to other industries as Web services for business integration are deployed. Web services will ultimately need to express and negotiate the policies under which such transactions are made.

Loosely Coupled Transactions

The flexibility offered by Web services for business process transactions far exceeds what can be accomplished using traditional 2-PC technologies. However, the more loosely we couple systems—separating them in time, space, and control—the more difficult it becomes to manage transactions distributed among them. Loosely coupled transactions, it would seem, come at a cost of increased complexity. That's true, but only so long as we keep trying to apply, refine, and improve traditional approaches based on 2-PC and ACID-style concepts. Instead, let's consider how to build an all-new transactional system based on loosely coupled Web services technologies: asynchronous communications, reliable messaging, and document-style interaction. Consider an example of a tightly-coupled system and how it can be improved.

Consider a person listening to the radio in her car, when she hears an announcement that a favorite musician will be performing in a nearby concert hall. She grabs her cell phone and dials the ticket-sales agency. A friendly salesperson answers the phone, and the driver launches her request—only to be interrupted by the salesperson, "I'm sorry, but our computers are down right now, and we don't know when they'll be back up. You'll have to call again later."

This demonstrates one of the drawbacks of synchronous transactions: In this case, there is nothing the driver can do but abort the transaction. The driver (requestor) and the reservation system (the provider) must be available simultaneously. There's no point for the driver to leave information with a salesperson who is just an intermediary, with no store-and-forward capability. Even if the salesperson were willing to take down the driver’s information, would the driver trust that person to complete the order? The responsibility for recovering from the system failure and restarting the transaction falls entirely on the requestor.

Half an hour later, the tries calling again and learns that the system is now available. Of course, the context of the driver’s transaction has been lost, so she has to start from the

very beginning. As luck would have it, the agent submits the driver’s request only to report, "Sorry, but all the concert seats are now sold out. The best I can do is row J, seats 103 and 104 in the upper mezzanine." For a period of a few minutes, the reservation system locks the database records that represent those two seats while the driver makes up his or her mind. If other customers are placing orders through different agents, they will not be offered the same seats. (This is now a synchronous transaction.)

The driver tells the agent she will take the tickets, but the driver’s cell phone dies just as she is about to jot down the confirmation number. Now what? Did the transaction complete? Does the driver really have two tickets for the concert, or does she need to call back and place another order? If so, will the driver end up with four tickets instead of two? Unfortunately, there's no way to know. Such are the problems of tightly coupled transactions without a reliable asynchronous messaging infrastructure.

Consider in this case, if the driver could just leave a voice-mail message (a self-contained document) including not only the obvious details, but instructions (the business logic) for what to do in case her first choice of seats isn't available? The voice mail message would then enter a message queue along with those of other customers and be processed in the sequence they are received. As a result of the driver’s request, the ticket agency would call her back or send an email message confirming the purchase. The acknowledgement would complete this long-lived, loosely coupled asynchronous transaction.

Long-lived transactions

By communicating asynchronously, the driver eliminated the real-time constraint of the transaction. The driver can make the request in the middle of the night. Even if a human agent must review the order, the driver need not be available at the time she submitted it. Although the vendor's voice-mail system must be able to accept calls at a reasonable rate, the actual transaction system that processes the request is highly scalable. Even if the transaction system goes offline, all orders will get processed in due time as long as customers can submit voice mail orders. This demonstrates how a reliable asynchronous messaging system is instrumental long-lived, loosely coupled asynchronous transactions.

Locking

Unlike 2-PC, this process can be implemented without the need for record locking, as long as all requests are submitted through a single queue and the ticket requests are processed serially. However, to achieve scalability, the application will need to introduce locking so the queue can be processed in parallel by several agents.

Compensating Transactions

Once a transaction has been committed, it can no longer be aborted. Yet in the real world, there are often times when the effects of a transaction must be undone. The problem is that some transactions cannot be reversed because their effects are permanent, and/or conditions have changed so much over time that restoring the

previous state would be inappropriate. As an example, consider a transaction that triggers the manufacturing of an item. Materials are consumed and money is spent. It is impossible to wipe out the transaction. You cannot un-manufacture the item. Instead, other actions must be taken, such as charging the customer a cancellation fee and offering the item for sale to other parties. In the earlier example of an unscrupulous travel aggregator who cancelled airline tickets at the last minute, we saw how the airline chose to put those seats on sale at a discount in order to make sure they would be sold and the airplane would be full.

These are examples of compensating transactions that can be applied after an original transaction has been committed in order to undo its effects, without necessarily returning resources to their original states. Many manual transaction processes support compensating transactions. In the case of long-lived, loosely coupled asynchronous Web services, compensating transactions can actually be used instead of resource locking.

Optimism

As noted earlier, 2-PC transactions are optimistic and assume a high likelihood of success. Imagine a human coordinator of a simple two-phase commit transaction commanding the participants. Phase One: "Okay, here's what you need to do. [Coordinator enumerates the requirements.] Has everyone prepared for the transaction by safely storing the results? Good." Phase Two, after receiving affirmative votes from all participants: "Now everyone...GO!" There's no need for the coordinator to ask whether anyone was unsuccessful, since all of the participants promised in Phase One that they could do as requested. The key to the success (and integrity) of the transaction is the locking of the resources between these two phases.

On the other hand, a loosely coupled transaction coordinator must take a pessimistic view of a transaction's outcome. Even with a reliable messaging protocol, many other errors can occur due to the long-term nature of the transaction. As noted earlier, even 2-PC transactions are subject to failure and require manual intervention. Rather than reserve their resources in advance, loosely coupled participants prepare compensating transactions that will undo the local effects in case the first phase is unsuccessful. If the transaction is later aborted, all participants execute their compensating transactions.

When using compensating transactions, a human coordinator might say in Phase One, "Okay, here's what you need to do. Don't do it yet, but in case this doesn't work, I want each of you to figure out ahead of time how to recover. Now everyone...GO!" Then, in Phase Two: "Great...did that work for everyone, or do we all need to run our back-out scenarios?"

Compensating transactions is one of the technologies that decouple systems from one another, and are a first step towards filling in the missing pieces of complex Web services.

Standards

IBM, Microsoft, and BEA are at work on WS-Coordination, a framework that supports multiple coordination types including WS-Atomic Transactions for short-lived "all-or-nothing" transactions (similar to 2-PC in nature), and WS-Business Activity for long-lived loosely coupled transactions using compensation.

Sun, Oracle, Iona, and others have announced plans for WS-CAF, the Web Services Composite Application Framework, for transactions and coordination of interdependent Web services.

The OASIS Business Transaction Technical Committee is continuing to develop the Business Transaction Protocol (BTP), but is awaiting implementations so progress can be made towards a full OASIS standard.

The issues are both political and technical. Because the traditional mechanisms for handling distributed transactions do not work for Web services, the standards for Web service transactions will be some of the last to be developed, agreed upon, and adopted. Most experts do not expect much impact from these competing standardization efforts until 2005. Therefore, in the interim, those faced with developing solutions for business integration are probably best advised to focus on the first two items in the list under section 2.3.2, “Cost-Risk Factors in Choosing Recovery Mechanisms.”

2.3.4. Are Web Services Distributed Transactions?

Based on the previous section, some may be left with the impression that Web services are a cowardly way out of distributed transactions. On the contrary, Web services offer alternative ways to perform distributed transactions, but do not eliminate the need for distributed transactions. When we deal with more than one resource manager, no matter what mechanism(s) we may use to accomplish the act of coordinating their activities, we *are* dealing with distributed transactions. In the previous section, the “Locking,” “Compensating Transactions,” and “Optimism,” sections discussed some alternative approaches to dealing with the distributed nature of long-lived and loosely-coupled transactions. The observation was made that locking is still required to obtain scalability in de-queuing client requests. Although the requests and responses are decoupled and become asynchronous, requirements for high throughput and scalability will dictate using some of the conventional mechanisms of distributed computing systems.

3. Appendix

3.1. Java Transaction Management

Note: It is assumed that the reader has some familiarity with J2EE terminology and architecture. If you are completely unfamiliar with these, suggested reading in the Resources Appendix may be useful to familiarize yourself before reading this section.

The Java Transaction Service (JTS) is a *component transaction monitor*. What does that mean? The concept of a *transaction processing monitor* (TPM) was introduced at the beginning of Section 2.0. It is a program that coordinates the execution of distributed transactions on behalf of an application. TPMs have been around for almost as long as databases; IBM first developed CICS, which is still used today, in the late 1960s. Classic (or *procedural*) TPMs manage transactions defined procedurally as sequences of operations on transactional resources (almost always databases). With the advent of distributed object protocols, such as CORBA, DCOM, and RMI, a more object-oriented view of transactions became desirable. Imparting transactional semantics to object-oriented components required an extension of the TPM model, in which transactions are instead defined in terms of invoking methods on transactional objects. JTS is just that: a component transaction monitor (sometimes called an *object transaction monitor, OTM*), or CTM.

At first glance, the transition from procedural transaction monitors to CTMs seems to be only a change in terminology. However, the difference is more significant. When a transaction in a CTM commits or rolls back, all the changes made by the objects involved in the transaction are either committed or undone as a group. But how does a CTM know what the objects did during that transaction? Transactional components like EJB components don't have commit() or rollback() methods, nor do they register what they've done with the transaction monitor. So how do the actions performed by J2EE components become part of the transaction?

The design of JTS and J2EE's transaction support was heavily influenced by the CORBA Object Transaction Service (OTS). In fact, many CORBA interfaces are implemented within the J2EE transactional framework. Using OTS instead of inventing a new object transaction protocol builds upon existing standards and opens the way for compatibility between J2EE and CORBA components. Another important part of the J2EE transaction framework is the Java Transaction API (JTA). J2EE Application Server programmers write code to implement the JTA, which controls low-level access to resource managers such as databases. JTS uses the JTA to do much of the low-level work of distributed object transactions. In fact, J2EE’s transactional framework is very complex and hard to understand from the “nuts-and-bolts” perspective. Fortunately, application programmers do not have to understand all the intricate details of how the JTS works to manage distributed transactions. There are many good documents available on how to program transactions using JTS, but no documents that really explain all the components of JTS in terms that can be understood from the familiar technology of TPMs.

There is nothing really mysterious about CORBA or JTS. The underlying mechanisms are very similar to TPMs. CORBA and JTS are component and object-oriented implementations of similar concepts that appear to add more complexity. In fact, this is somewhat true, but the added complexity is necessary to derive a model that is flexible and extensible in a highly distributed

programming environment. Sun Microsystems, Inc. does a good job of introducing the basic components and their relationships in a few sentences:

> **Java Transaction Service (JTS)** specifies the implementation of a Transaction Manager which supports the Java Transaction API (JTA) 1.0 Specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification at the low-level. JTS uses the standard CORBA ORB/TS interfaces and Internet Inter-ORB Protocol (IIOP) for transaction context propagation between JTS Transaction Managers.
>
> A JTS Transaction Manager provides transaction services to the parties involved in distributed transactions: the application server, the resource manager, the standalone transactional application, and the Communication Resource Manager (CRM). [3]

Sun Microsystems followed the Distributed Transaction Processing (DTP) model of the OpenGroup organization (www.opengroup.org). The OpenGroup specifications define major components participating in the DTP model as well as a set of APIs that define communication between these components. Components participating in the DTP model are: application programs, resource managers, and a transaction manager. The interface defined between application programs wishing to participate in global transactions and the transaction manager is called the TX interface, while the interface between transaction managers and the resource managers is called the XA interface (described in Section 2.1.4.) The JTS and JTA specifications describe the workings of DTP within the J2EE transaction framework.

JTS defines several major players in the DTP model of Java Enterprise middleware:

- **JTS Transaction Manager**: A core component that provides services of transaction resource management (i.e., resource enlistment, de-enlistment), transaction demarcation, synchronization notification callbacks, transaction context propagation, and two-phase commit initiation and recovery coordination with resource managers.
- **Application Server**: provides infrastructure required to support the application run-time environment, called *Containers* in J2EE terminology.
- **Resource Manager**: A component that manages access to a persistent stable storage system. Each resource manager cooperates with a transaction manager in commit initiation and failure recovery. An example of resource manager would be a database server.
- **Resource Adapter**: This is pluggable interface that provides standard access to resource managers. By using the concept of a resource adapter, access to resource managers can be abstracted and configured without declaratively in and administration tools such as the SAP Visual Admin Tool.
- **J2EE Application**: Applications can manage transactions from java programs running outside of a J2EE Application Server or J2EE Applications can run within the context of a J2EE Container and invoke transactions from EJBs, JSPs and native Servlets. The EJB Container has special features to simply transaction management using either declarative transactional semantics within deployment descriptors (the

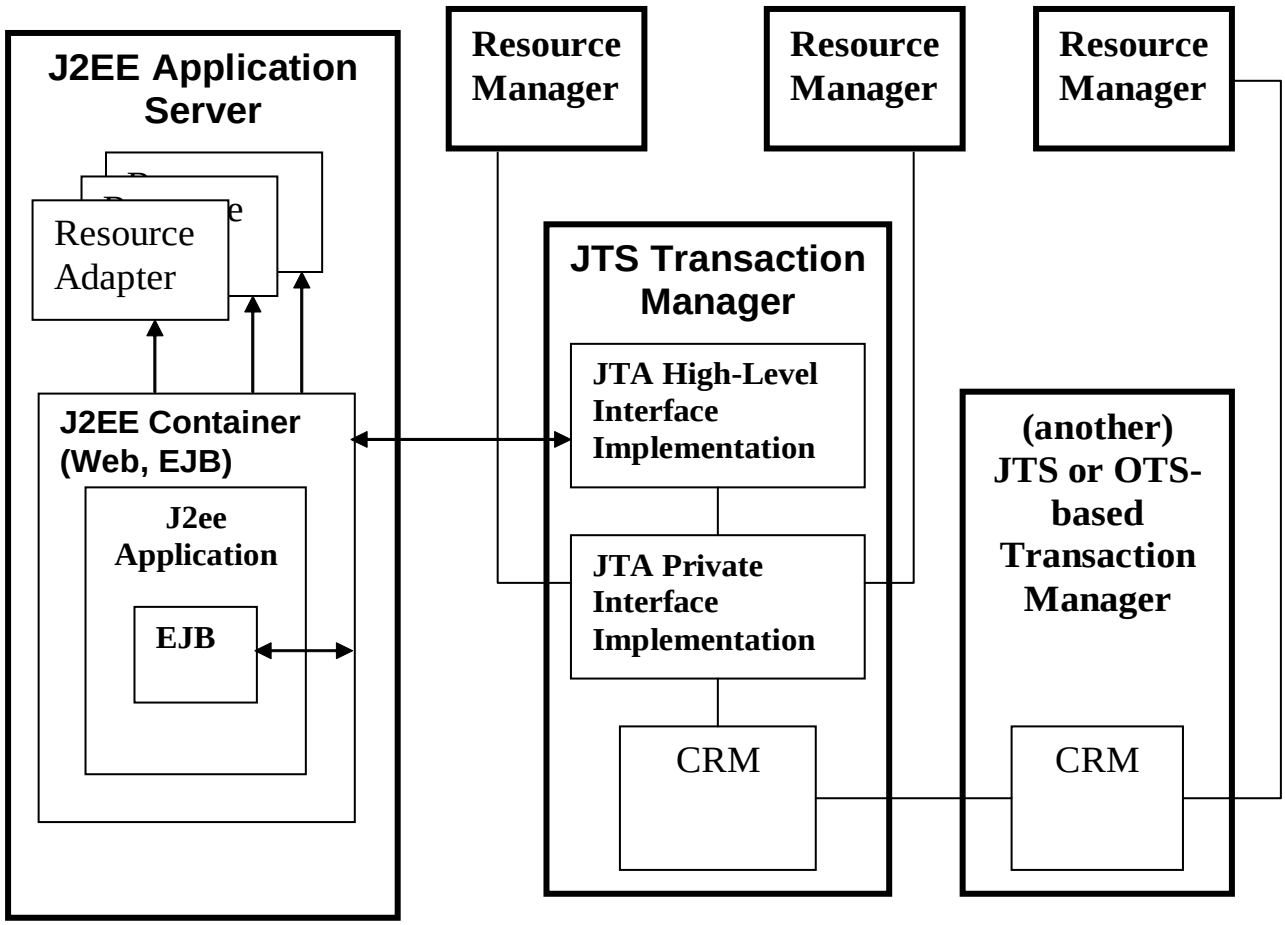
[3] From Java Transactions Download Web page: http://java.sun.com/j2ee/transactions/downloads/

ejb-jar.jar xml descriptor) or programmatic transaction management using the `JTS` interfaces. Normally, transactional applications take advantage of the features of a J2EE Container but the JTS specification does not impose any limitations on this.

- **Communication Resource Manager (CRM)**: This component supports transactional context propagation. Simply put, this component allows the transaction manager to participate in transactions initiated by other transaction managers. JTS does not specify a protocol for this component.

The Java Transaction API consists of three elements: a high-level transaction demarcation interface intended for use by J2EE Applications (including container-managed applications), a high-level transaction manager interface intended for use by application servers, and a Java mapping of the X/Open XA protocol intended for resource managers.

The diagram below shows the relationship between key players in J2EE transaction management:



In the diagram above, note that the EJB component communicates with the EJB container and not directly with the transaction manager. The EJB container in turn communicates with the transaction manager on behalf of the EJB and other components requiring transaction support. The J2EE server uses resource adapters to obtain JTA objects that in turn know how to connect to their resource managers (there is a one-to-one correspondence between resource adapters and resource managers.) The J2EE Server passes these objects to the transaction manager which in turn manages the actual transaction process. The implementation of the low-level (private) JTA interface manages access to the tesource managers on behalf of the J2EE server with the help of the CRM if other transaction managers are involved in the transaction.

Although the diagram above is much more complex than the one in 2-PC landscape diagram in Section 2.12, the overall structure is similar. A transaction manager called by an application invoking a 2-PC transaction takes the role of a transaction coordinator on the given node where the transaction manager was called. This coordinator enlists the help of transaction managers on other nodes involved in the distributed transaction. The transaction managers in turn communicate with resource managers to carry out the individual transactions.

Transparent Resource Enlistment

While the application state is manipulated by components, it is still stored in transactional resource managers (for example, databases and message queue servers), which can be registered as resource managers in a distributed transaction. Section 2, discusses how multiple resource managers could be enlisted in a single transaction, coordinated by a transaction manager. Resource managers know how to associate changes in application state with specific transactions.

But this just moves the focus from the component to the resource manager—how does the J2EE server figure out what resources are involved in the transaction so it can enlist them? Consider the following code, which might be found in a typical EJB session bean:

Listing 1. Transparent resource enlistment with bean-managed transactions

```
InitialContext ic = new InitialContext();
UserTransaction ut = ejbContext.getUserTransaction();
ut.begin();

DataSource db1 = (DataSource) ic.lookup("java:comp/env/OrdersDB");
DataSource db2 = (DataSource) ic.lookup("java:comp/env/InventoryDB");
Connection con1 = db1.getConnection();
Connection con2 = db2.getConnection();
// perform updates to OrdersDB using connection con1
// perform updates to InventoryDB using connection con2
ut.commit();
```

Notice that there is no code in this example to enlist the JDBC connections in the current transaction—the J2EE Server does this for us.

Three Types of Resource Managers

When an EJB component wants to access a database, a message queue server, or some other transactional resource, it can acquire a connection to the resource manager (usually by using the Java Naming and Directory Interface (JNDI.) Moreover, the J2EE specification only recognizes

three types of transactional resources—JDBC databases, JMS message queue servers, and "other transactional services accessed through JCA." Services in the latter class (such as ERP systems) must be accessed through JCA (the J2EE Connector Architecture). For each of these types of resources, either the J2EE container or the provider helps to enlist the resource into the transaction. Also, the J2EE specification has special classes of EJBs that are managed entirely by the EJB container. For Container Managed Beans and Message-Driven Beans, the EJB developer must leave all database access management to the EJB container.

In Listing 1 above, `con1` and `con2` appear to be ordinary JDBC connections such as those that would be returned from `DriverManager.getConnection()`. We get these connections from a JDBC `DataSource`, which was obtained by looking up the name of the data source in JNDI. The name used in our EJB component to find the data source (`java:comp/env/OrdersDB`) is specific to the component; the `resource-ref` section of the component's deployment descriptor maps it to the JNDI name of some application-wide `DataSource` managed by the J2EE container.

The Hidden JDBC Driver

Every J2EE container can create transaction-aware pooled `DataSource` objects, but the J2EE specification doesn't show you how, because it's outside the spec. If you browse the J2EE documentation, you won't find anything on how to create JDBC data sources. You'll have to look in the documentation for your J2EE container instead. Depending on your J2EE container, creating a data source might involve adding a data source definition to a property or configuration file, or might be done through a GUI administration tool. In the case of the SAP J2EE Engine, the Visual Administrator Tool is the easiest way to configure DataSources and associated JDBC drivers.

Each J2EE container (or connection pool manager, like PoolMan) provides its own mechanism for creating a `DataSource`, and it is in this mechanism that the JTA magic is hidden. The connection pool manager obtains a `Connection` from the specified JDBC driver, but before returning it to the application, wraps it with a facade that also implements `Connection`, interposing itself between the application and the underlying connection. When the connection is created or a JDBC operation is performed, the wrapper asks the transaction manager if the current thread is executing in the context of a transaction, and automatically enlists the `Connection` in the transaction if one exists.

The other types of transactional resources, JMS message queues and JCA connectors, rely on a similar mechanism to hide resource enlistment from the user. When you make a JMS queue available to a J2EE application at deployment time, you again use a provider-specific mechanism to create the managed JMS objects (queue connection factories and destinations), which you then publish in a JNDI namespace. The managed objects created by the provider contain similar auto-enlistment code as the JDBC wrapper added by the J2EE container-supplied connection pool manager.

Transparent Transaction Control

The two types of J2EE transactions—container-managed and bean-managed—differ in how they start and end a transaction. Where a transaction starts and ends is referred to as *transaction demarcation*. The example code in Listing 1 demonstrates a bean-managed transaction (sometimes called a *programmatic* transaction.) Bean-managed transactions are started and ended

explicitly by a component using the `UserTransaction` class. `UserTransaction` is made available to EJB components through the `ejbContext` and to other J2EE components through JNDI.

Container-managed transactions (or *declarative transactions*) are started and ended transparently on behalf of the application by the J2EE container, based on transaction attributes in the component's deployment descriptor. You indicate whether an EJB component uses bean-managed or container-managed transactional support by setting the `transaction-type` attribute to either `Container` or `Bean`.

With container-managed transactions, you can assign transactional attributes at either the EJB class or method levels; you can specify a default set of transactional attributes for the EJB class, and you can also specify attributes for each method if different methods are to have different transactional semantics. These transactional attributes are specified in the `container-transaction` section of the assembly descriptor. An example assembly descriptor is shown in Listing 2. The supported values for the `trans-attribute` are:

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

The `trans-attribute` determines if the method supports execution within a transaction, what action the J2EE container should take when the method is called within a transaction, and what action the J2EE container should take if it is called outside of a transaction. The most common container-managed transaction attribute is `Required`. When `Required` is set, a transaction in process will enlist your bean in that transaction, but if no transaction is running, the J2EE Container will start one for you. We will investigate the differences between the various transaction attributes, and when you might want to use each, in Part 3 of this series.

Listing 2. Sample EJB assembly descriptor

```
<assembly-descriptor>
  ...
  <container-transaction>
    <method>
      <ejb-name>MyBean</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>MyBean</ejb-name>
      <method-name>updateName</method-name>
      </method>
   <trans-attribute>RequiresNew</trans-attribute>
  </container-transaction>
  ...
</assembly-descriptor>
```

Declarative Transaction Management

Unlike the example in Listing 1, with declarative transaction demarcation there is no transaction management code in the component methods. Not only does this make the resulting component code easier to read (because it is not cluttered with transaction management code), but it has another, more significant advantage—the transactional semantics of the component can be changed at application assembly time, without modifying or even accessing the source code for the component.

While being able to specify transaction demarcation separate from the code is a very powerful feature, making poor decisions at assembly time can render your application unstable or seriously impair its performance. The responsibility for correctly demarcating container-managed transactions is shared between the component developer and the application assembler. The component developer needs to provide sufficient documentation as to what the component does, so that the application deployer can make intelligent decisions on how to structure the application's transactions. The application assembler needs to understand how the components in the application interact, so that transactions can be demarcated in a way that enforces application consistency and doesn't impair performance.

Transparent Transaction Propagation

In either type of transaction, resource enlistment is transparent; the J2EE container automatically enlists any transactional resources used during the course of the transaction into the current transaction. This process extends not only to resources used by the transactional method, such as the database connections acquired in Listing 1, but also by methods it calls—even remote methods. Let's take a look at how this happens.

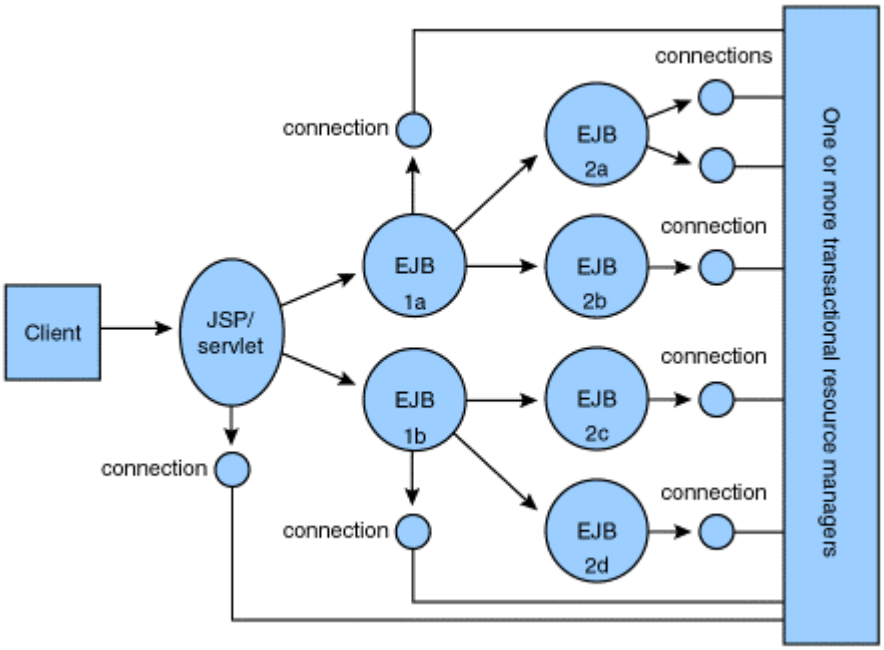
Let's say that methodA() of object A starts a transaction, and then calls methodB() of object B, which acquires a JDBC connection and updates the database. The connection acquired by B will be automatically enlisted in the transaction created by A. How did the J2EE container know to do this?

When a transaction is initiated, the transaction context is associated with the executing thread. When A creates the transaction, the thread in which A is executing is associated with that transaction. Because local method invocations execute in the same thread as the caller, any methods called by A will also be in the context of that transaction.

What if object B is really a stub to an EJB component executing in another thread or even another JVM? Amazingly, resources accessed by remote object B will still be enlisted in the current transaction. The EJB object stub (the part that executes in the context of the caller), the EJB protocol (RMI over IIOP), and the skeleton object on the remote end all conspire to make this happen transparently. The stub determines if the caller is executing a transaction. If so, the transaction ID, or Xid, is propagated to the remote object as part of the IIOP call along with the method parameters. (IIOP is the CORBA remote-invocation protocol, which provides for propagating various elements of execution context, such as transaction context and security context. See the Resources Appendix for more information on RMI over IIOP.) If the call is part of a transaction, the skeleton object on the remote system automatically sets the remote thread's transaction context, so that when the actual remote method is invoked, it is already part of the

transaction. (The stub and skeleton objects also take care of beginning and committing J2EE container-managed transactions.)

Transactions can be initiated by any J2EE component—an EJB component, a servlet, or a JSP page (or an application client, if the J2EE container supports it.)[4] This means that your application can start a transaction in a servlet or JSP page when a request arrives, do some processing within the servlet or JSP page, access entity beans and session beans on multiple servers as part of the page's logic, and have all of this work be part of one transaction, transparently. The figure below demonstrates how the transaction context follows the path of execution from servlet to EJB to EJB.



[4] Note, the SAP J2EE Engine does not support transactions initiated from a application clients. This is an optionally supported feature in J2EE 1.4.

3.2. Resources

- The *Java theory and practice* column archive includes Part 1 of this series: "An introduction to transactions."
- *Transaction Processing: Concepts and Techniques* by Jim Grey and Andreas Reuter is the definitive work on the subject of transaction processing.
- *Principles of Transaction Processing* by Philip Bernstein and Eric Newcomer is a fine introduction to the subject; it covers a lot of history as well as concepts.
- The Java Transaction Service specification is quite readable and offers a high-level explanation of how an object transaction monitor fits into distributed applications.
- JTS was built on top of the existing CORBA OTS specification.
- The lower-level details of transactional support in J2EE is detailed in the Java Transaction API specification.
- The J2EE specification describes how JTS and JTA fit into J2EE and how transactions interact with other J2EE technologies, such as Enterprise JavaBeans technology.
- Damian Hagge provides an excellent introduction to RMI over IIOP in his article "RMI-IIOP in the enterprise" (*developerWorks*, March 2002).
- Interested in building J2EE applications using IBM tools? Check out the IBM Redbook *Programming J2EE APIs with WebSphere Advanced Edition* .
- The VisualAge Developer Domain hosts an article that examines database access and concurrency management issues in J2EE applications.
- "Optimistic locking pattern for EJBs" explores the practice of optimistic locking, which can make transaction processing more efficient.
- The IBM Redbook *Building Enterprise Web Transactions using VisualAge Generator JavaBeans and JSPs* explores automated generation of JSP pages which can access enterprise data transactionally.
- The IBM Redbook *Enterprise JavaBeans Development Using VisualAge for Java* describes IBM's rapid application development support for EJB applications.
- "Transaction management under J2EE 1.2" by Sanjay Mahapatra covers the basics of declarative transaction demarcation.
- Find other Java technology resources on the *developerWorks* Java technology zone.
- "Oracle Guide to Distributed Transaction", Copyright ©1996, Oracle Corporation All rights reserved.