JBoss Transactions API 4.8.0

Programmers Guide

JBTA-PG-9/8/09

Legal Notices

The information contained in this documentation is subject to change without notice.

JBoss Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. JBoss Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Arjuna is a trademark of Hewlett-Packard Company and used here under licence.

Copyright

Software Version

JBoss Transactions API 4.8.0

Restricted Rights Legend

Contents

Table Of Contents

About This Guide

What This Guide Contains

This document describes how to use the Arjuna Technologies JTA and JDBC implementations, referred to within this document as JBoss Transactions API 4.8.0 (JBossJTA).

Audience

This guide is most relevant to engineers who are responsible for administering JBoss Transactions API 4.8.0 installations.

Prerequisites

Knowledge of JTA and JDBC.

Organization

This guide contains the following chapters:

- **Chapter 1, An introduction to the JTA:** An introduction to JTA by describing the concept of the high level interface to manage transactions.
- **Chapter 2, Transactions:** contains an overview of the interfaces defined by JTA demarcate and manage transactions from the user view and from the application server view.
- **Chapter 3, The Resource Manager:** gives an overview of the way to manage Resource Managers compliant to the X/Open XA protocol.
- **Chapter 4, Transaction Recovery:** This chapter, gives an overview of the behavior applied to recovery XA resource managers, and describes information needed by JBossJTA to manage the recovery.
- **Chapter 5, JDBC and transaction:** This chapter, describes how JBossJTA supports transactional applications that access databases using JDBC.

- **Chapter 6, Example:** illustrates how to build transactional applications with JBossJTA- accessing a database with JDBC and managing recovery.
- **Chapter 7, Configuring JBossJTA:** describes how to configure JBossJTA features.
- **Chapter 8, Using JBossJTA in the application servers:** describes how to use and configure JBossJTA when embedded in an application server, particularly JBoss.

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
`Code`	Text that represents programming code.
Function | Function	A path to a function or dialog box within an interface. For example, "Select File | Open." indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
Note: **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss Transactions API 4.8.0 documentation set:

- **JBoss Transactions API 4.8.0** *Release Notes*: Provides late-breaking information about JBoss Transactions API 4.8.0.
- **JBoss Transactions API 4.8.0** *Installation Guide*: This guide provides instructions for installing JBoss Transactions API 4.8.0.

- **JBoss Transactions API 4.8.0** *Administrators Guide*: Provides guidance for administering the system.

Contacting Us

Questions or comments about JBoss Transactions API 4.8.0should be directed to our support team.

An introduction to the JTA

The Java Transaction API

The interfaces specified by the many transaction standards are typically too low-level for most application programmers. Therefore, Sun Microsystems has specified higher-level interfaces to assist in the development of distributed transactional applications. Note, these interfaces are still low-level, and require, for example, the programmer to be concerned with state management and concurrency for transactional application. In addition, they are geared more for applications which require XA resource integration capabilities, rather than the more general resources which the other APIs allow.

With reference to [JTA99], distributed transaction services typically involve a number of participants:

- *application server*: which provides the infrastructure required to support the application run-time environment which includes transaction state management, e.g., an EJB server.
- *transaction manager*: provides the services and management functions required to support transaction demarcation, transactional resource management, synchronisation and transaction context propagation.
- *resource manager*: (through a resource adapter[1]) provides the application with access to resources. The resource manager participates in distributed transactions by implementing a transaction resource interface used by the transaction manager to communicate transaction association, transaction completion and recovery.
- *a communication resource manager (CRM)*: supports transaction context propagation and access to the transaction service for incoming and outgoing requests.

From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed; only high-level interfaces need to be defined to allow transaction demarcation, resource enlistment, synchronization and recovery process to be driven from the users of the transaction services. The JTA is a high-level application interface that allows a transactional application to demarcate transaction boundaries, and contains also contains a mapping of the X/Open XA protocol.

Note: the JTA support provided by *JBossJTA* is compliant with the 1.1 specification.

[1] A Resource Adapter is used by an application server or client to connect to a Resource Manager. JDBC drivers which are used to connect to relational databases are examples of Resource Adapters.

Chapter 2

Transactions

The API

The Java Transaction API consists of three elements: a high-level application transaction demarcation interface, a high-level transaction manager interface intended for application server, and a standard Java mapping of the X/Open XA protocol intended for transactional resource manager. All of the JTA classes and interfaces occur within the `javax.transaction` package, and the corresponding *JBossJTA* implementations within the `com.arjuna.ats.jta` package.

Caution: Each Xid that JBossTS creates must have a unique node identifier encoded within it and JBossTS will only recover transactions and states that match a specified node identifier. The node identifier to use should be provided to JBossTS via the CoreEnvironmentBean.`nodeIdentifier` property. You must make sure this value is unique across your JBossTS instances. If you do not provide a value, then JBossTS will fabricate one and report the value via the logging infrastructure. The contents of this should be alphanumeric and not exceed 64 bytes in length.

UserTransaction

The `UserTransaction` interface provides applications with the ability to control transaction boundaries. It has methods for beginning, committing, and rolling back top-level transactions: nested transactions are not supported, and `begin` throws the `NotSupportedException` when the calling thread is already associated with a transaction. `UserTransaction` automatically associates newly created transactions with the invoking thread.

Note: In *JBossJTA*, UserTransactions can be obtained from the static com.arjuna.ats.jta.UserTransaction.userTransaction() method.

In order to select the local JTA implementation it is necessary to perform the following steps:

1. make sure the property JTAEnvironmentBean.jtaTMImplementation is set to com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple.
2. make sure the property JTAEnvironmentBean.jtaUTImplementation is set to com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple.

TransactionManager

The `TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed.

> **Note:** In *JBossJTA*, transaction manager implementations can be obtained from the static `com.arjuna.ats.jta.TransactionManager.transactionManager` method.

The Transaction Manager maintains the transaction context association with threads as part of its internal data structure. A thread's transaction context is either *null* or it refers to a specific global transaction. Multiple threads may be associated with the same global transaction. As noted above, nested transactions are not supported.

Each transaction context is encapsulated by a `Transaction` object, which can be used to perform operations which are specific to the target transaction, regardless of the calling thread's transaction context.

The `begin` method of `TransactionManager` starts a new top-level transaction and associates the transaction context with the calling thread. If the calling thread is already associated with a transaction then it throws the `NotSupportedException`.

The `getTransaction` method returns the `Transaction` object that represents the transaction context currently associated with the calling thread. This object can be used to perform various operations on the target transaction, described later.

The `commit` method is used to complete the transaction currently associated with the calling thread. After it returns, the calling thread is associated with no transaction. If `commit` is called when the thread is not associated with any transaction context, the TM throws an exception. In some implementation, the commit operation is restricted to the transaction originator only. If the calling thread is not allowed to commit the transaction, the TM throws an exception. *JBossJTA* does not currently impose any restriction on the ability of threads to terminate transactions.

The `rollback` method is used to rollback the transaction associated with the current thread. After the `rollback` method completes, the thread is associated with no transaction.

> **Note:** In a multi-threaded environment it is possible that multiple threads are active within the same transaction. If checked transaction semantics have been disabled, or the transaction times out, then it is possible for a transaction to be terminated by a thread other than the one that created it. In this case, it is often important that this information is communicated to the creator. JBossTS does this during `commit` or `rollback` by throwing `IllegalStateException`.

Suspending and resuming a transaction

The JTA supports the concept of a thread temporarily suspending and resuming transactions to enable it to perform non-transactional work. The `suspend` method is called to temporarily suspend the current transaction that is associated with the calling thread, i.e., so that the thread is no longer operating within its scope. If the thread is not associated with any transaction, a `null` object reference is returned; otherwise, a valid `Transaction` object is returned. The `Transaction` object can later be passed to the `resume` method to reinstate the transaction context.

The `resume` method associates the specified transaction context with the calling thread. If the transaction specified is a valid transaction, the transaction context is associated with the calling thread; otherwise, the thread is associated with no transaction.

> **Note:** if `resume` is invoked when the calling thread is already associated with another transaction, the Transaction Manager throws the `IllegalStateException` exception.

```
Transaction tobj = TransactionManager.suspend();
..
TransactionManager.resume(tobj);
```

> **Note:** some transaction manager implementations allow a suspended transaction to be resumed by a different thread. This feature is not required by JTA, but *JBossJTA* does support this.

When a transaction is suspended the application server must ensure that the resources in use by the application are no longer registered with the suspended transaction. When a resource is de-listed this triggers the Transaction Manager to inform the resource manager to disassociate the transaction from the specified resource object. When the application's transaction context is resumed, the application server must ensure that the resources in use by the application are again enlisted with the transaction. Enlisting a resource as a result of resuming a transaction triggers the Transaction Manager to inform the resource manager to re-associate the resource object with the resumed transaction.

The Transaction interface

The `Transaction` interface allows operations to be performed on the transaction associated with the target object. Every top-level transaction is associated with one `Transaction` object when the transaction is created. The `Transaction` object can be used to:

- enlist the transactional resources in use by the application.
- register for transaction synchronization call backs.
- commit or rollback the transaction.
- obtain the status of the transaction.

The `commit` and `rollback` methods allow the target object to be committed or rolled back. The calling thread is not required to have the same transaction associated with the thread. If the calling thread is not allowed to commit the transaction, the transaction manager

throws an exception. At present *JBossJTA* does not impose restrictions on threads terminating transactions.

> **Note:** The JTA standard does not provide a means to obtain the transaction identifier. However, in *JBossJTA* there are several ways in which this can be viewed. Simply printing the transaction instance (calling `toString`) will give full information about the transaction, containing the identifier. Alternatively you can cast the `javax.transaction.Transaction` instance to a `com.arjuna.ats.jta.transaction.Transaction` and then call `get_uid`, which returns an ArjunaCore `Uid` representation, or `getTxId`, which returns an `Xid` for the global identifier, i.e., no branch qualifier.

Resource enlistment

Transactional resources such as database connections are typically managed by the application server in conjunction with some resource adapter and optionally with connection pooling optimization. In order for an external transaction manager to co-ordinate transactional work performed by the resource managers, the application server must enlist and de-list the resources used in the transaction. These resources (participants) are enlisted with the transaction so that they can be informed when the transaction terminates, e.g., are driven through the two-phase commit protocol.

As stated previously, the JTA is much more closely integrated with the XA concept of resources than the arbitrary objects. For each resource in-use by the application, the application server invokes the `enlistResource` method with an `XAResource` object which identifies the resource in use. See for details on how the implementation of the `XAResource` can affect recovery in the event of a failure.

The enlistment request results in the transaction manager informing the resource manager to start associating the transaction with the work performed through the corresponding resource. The transaction manager is responsible for passing the appropriate flag in its `XAResource.start` method call to the resource manager.

The `delistResource` method is used to disassociate the specified resource from the transaction context in the target object. The application server invokes the method with the two parameters: the `XAResource` object that represents the resource, and a flag to indicate whether the operation is due to the transaction being suspended (`TMSUSPEND`), a portion of the work has failed (`TMFAIL`), or a normal resource release by the application (`TMSUCCESS`).

The de-list request results in the transaction manager informing the resource manager to end the association of the transaction with the target `XAResource`. The flag value allows the application server to indicate whether it intends to come back to the same resource whereby the resource states must be kept intact. The transaction manager passes the appropriate flag value in its `XAResource.end` method call to the underlying resource manager.

Transaction synchronization

Transaction synchronization allows the application server to be notified before and after the transaction completes. For each transaction started, the application server may optionally register a `Synchronization` call back object to be invoked by the transaction manager:

- The `beforeCompletion` method is called prior to the start of the two-phase transaction complete process. This call is executed in the same transaction context of the caller who initiates the `TransactionManager.commit` or the call is executed with no transaction context if `Transaction.commit` is used.
- The `afterCompletion` method is called after the transaction has completed. The status of the transaction is supplied in the parameter. This method is executed without a transaction context.

Transaction equality

The transaction manager implements the `Transaction` object's `equals` method to allow comparison between the target object and another `Transaction` object. The `equals` method should return *true* if the target object and the parameter object both refer to the same global transaction.

```
Transaction txObj = TransactionManager.getTransaction();
Transaction someOtherTxObj = ..
..

boolean isSame = txObj.equals(someOtherTxObj);
```

TransactionSynchronizationRegistry

The javax.transaction.TransactionSynchronizationRegistry interface, added to the JTA API in version 1.1, provides for registering Synchronizations with special ordering behavior and for storing key-value pairs in a per transaction Map. Full details are available from the JTA 1.1 API specification and javadoc. Here we focus on implementation specific behavior.

In standalone environments, the Implementation can be accessed as follows:

```
   javax.transaction.TransactionSynchronizationRegistry tsr = new
com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionSynchronizati
onRegistryImple();
```

This is a stateless object and hence is cheap to instantiate.

In application server environments the standard JNDI name binding is java:comp/TransactionSynchronizationRegistry

Ordering of interposed Synchronizations is relative to other local Synchronizations only. In cases where the transaction is distributed over multiple JVMs, global ordering is not guaranteed.

The per-transaction data storage provided by the TransactionSynchronizationRegistry getResource/putResource is non-persistent and thus not available in Transactions during crash recovery. When running integrated with an application server or other container, this storage may be used for system purposes. To avoid collisions it is recommended to use an application specific prefix on map keys. e.g. put("myapp_"+key, value). The behavior of the Map on Threads that have status NO_TRANSACTION or where the transaction they are associated with has been rolled back by another Thread, such as in the case of a timeout, is undefined. It is possible to have a Transaction be associated with multiple Threads. For such cases the Map is synchronized to provide thread safety.

Chapter 1

The Resource Manager

The XAResource interface

Whereas some transaction specifications and systems define a generic resource which can be used to register arbitrary resources with a transaction, the JTA is much more XA specific. The `javax.transaction.xa.XAResource` interface is a Java mapping of the XA interface. The `XAResource` interface defines the contract between a Resource Manager and a Transaction Manager in a distributed transaction processing environment. A resource adapter for a resource manager implements the `XAResource` interface to support association of a top-level transaction to a resource such as a relational database.

The `XAResource` interface can be supported by any transactional resource adapter that is intended to be used in an environment where transactions are controlled by an external transaction manager, e.g., a database management system. An application may access data through multiple database connections. Each database connection is associated with an `XAResource` object that serves as a proxy object to the underlying resource manager instance. The transaction manager obtains an `XAResource` for each resource manager participating in a top-level transaction. It uses the `start` method to associate the transaction with the resource, and it uses the `end` method to disassociate the transaction from the resource.

The resource manager is responsible for associating the transaction with all work performed on its data between the `start` and `end` invocations. At transaction commit time, these transactional resource managers are informed by the transaction manager to prepare, commit, or rollback the transaction according to the two-phase commit protocol.

In order to be better integrated with Java, the `XAResource` differs from the standard XA interface in the following ways:

- The resource manager initialization is done implicitly by the resource adapter when the resource (connection) is acquired. There is no `xa_open` equivalent.
- `Rmid` is not passed as an argument. Each `Rmid` is represented by a separate `XAResource` object.
- Asynchronous operations are not supported because Java supports multi-threaded processing and most databases do not support asynchronous operations.
- Error return values that are caused by the transaction manager's improper handling of the `XAResource` object are mapped to Java exceptions via the `XAException` class.
- The DTP concept of "Thread of Control" maps to all Java threads that are given access to the `XAResource` and `Connection` objects. For example, it is legal for

two different threads to perform the start and end operations on the same XAResource object.

Extended XAResource control

By default, whenever an XAResource object is registered with a JTA compliant transaction service, you have no control over the order in which it will be invoked during the two-phase commit protocol, with respect to other XAResource objects. In *JBossTS*, however, there is support for controlling the order via the two interfaces com.arjuna.ats.jta.resources.StartXAResource and com.arjuna.ats.jta.resources.EndXAResource. By inheriting your XAResource instance from either of these interfaces, you control whether an instance of your class will be invoked first or last, respectively.

Note: Only one instance of each interface type may be registered with a specific transaction.

In the TxCore manual we discussed the Last Resource Commit optimization (LRCO), whereby a single resource that is only one-phase aware (does not support prepare), can be enlisted with a transaction that is manipulating two-phase aware participants. This optimization is also supported within the JTA aspects of *JBossTS*.

In order to use the LRCO, your XAResource implementation must extend the com.arjuna.ats.jta.resources.LastResourceCommitOptimisation marker interface (it provides no methods). When enlisting the resource via Transaction.enlistResource, *JBossTS* will ensure that only a single instance of this type of participant is used within each transaction. Your resource will be driven last in the commit protocol: no invocation of prepare will occur.

Note: By default an attempt to enlist more than one instance of a LastResourceCommitOptimisation class will fail and *false* will be returned from Transaction.enlistResource. This behaviour can be overridden by setting the com.arjuna.ats.jta.allowMultipleLastResources to true. However, before doing so you should read the section on enlisting multiple one-phase aware resources.

In order to utilize the LRCO in a distributed environment, it is necessary to disable interposition support. It is still possible to use implicit context propagation.

Enlisting multiple one-phase aware resources

As discussed in the Transaction Core documentation, in order to guarantee consistency (atomicity) of outcome between multiple participants (resources) within the same transaction, the two-phase commit protocol is used with a durable transaction log. In the case of possessing a single one-phase aware resource, it is still possible to achieve an atomic (all of nothing) outcome across resources by utilizing the Last Resource Commit Optimization, as explained earlier.

However, there may be situations where multiple one-phase aware resources are enlisted within the same transaction. For example, a legacy database running within the same transaction as a legacy JMS implementation. In these situations it is not possible to achieve atomicity of transaction outcome across multiple resources because none of them enter the prepare (waiting for final outcome) state: they commit or rollback immediately when instructed by the transaction coordinator, without knowledge of other resource states and without any way of undoing should subsequent resources make a different choice. This can result in data corruption or heuristic outcomes.

In these situations we recommend one of the following approaches:

- Wrap the resources in compensating transactions. See the Web Services transactions guides for further details.
- Migrate the legacy implementations to two-phase aware equivalents.

In the cases where neither of these options are viable, JBossTS does support the enlistment of multiple one-phase aware resources within the same transaction. In order to do this, see the section on the Last Resource Commit Optimization.

Caution: Even when this support is enabled, JBossTS will issue warnings when it detects that the option has been enabled ("*You have chosen to enable multiple last resources in the transaction manager. This is transactionally unsafe and should not be relied upon.*") and when multiple one-phase resources are enlisted within the transaction ("*This is transactionally unsafe and should not be relied on.*").

Note: You can choose to override the above mentioned warning at runtime by setting the `com.arjuna.ats.jta.disableMultipleLastResourcesWarning` property to true. You will see a warning that you have done this when JBossTS starts up and see the warning about enlisting multiple one-phase resources once the first time this occurs, but after that no further warnings will be output. You should obviously only consider changing the default value of this property (false) with caution.

Opening a Resource Manager

The X/Open XA interface requires that the transaction manager initialize a resource manager (`xa_open`) prior to any other `xa_` calls. JTA requires initialization of a resource manager to be embedded within the resource adapter that represents the resource manager. The transaction manager does not need to know how to initialize a resource manager; it is only responsible for informing the resource manager about when to start and end work associated with a transaction and when to complete the transaction. The resource adapter is responsible for opening (initializing) the resource manager when the connection to the resource manager is established.

Closing a Resource Manager

A resource manager is closed by the resource adapter as a result of destroying the transactional resource. A transaction resource at the resource adapter level is comprised of two separate objects:

- An `XAResource` object that allows the transaction manager to start and end the transaction association with the resource in use and to coordinate transaction completion process.
- A connection object that allows the application to perform operations on the underlying resource (for example, JDBC operations on an RDBMS).

Once opened, the resource manager is kept open until the resource is released (closed) explicitly. When the application invokes the connection's `close` method, the resource adapter invalidates the connection object reference that was held by the application and notifies the application server about the close. The transaction manager should invoke the `XAResource.end` method to disassociate the transaction from that connection.

The `close` notification allows the application server to perform any necessary cleanup work and to mark the physical XA connection as free for reuse, if connection pooling is in place.

Threads of control

The X/Open XA interface specifies that the transaction association related `xa` calls must be invoked from the same thread context. This thread-of-control requirement is not applicable to the object-oriented component-based application run-time environment, in which application threads are dispatched dynamically at method invocation time. Different threads may be using the same connection resource to access the resource manager if the connection spans multiple method invocation. Depending on the implementation of the application server, different threads may be involved with the same `XAResource` object. The resource context and the transaction context may be operated independent of thread context. This means that it is possible for different threads to be invoking the `start` and `end` methods.

If the application server allows multiple threads to use a single `XAResource` object and the associated connection to the resource manager, it is the responsibility of the application server to ensure that there is only one transaction context associated with the resource at any point of time. Thus the `XAResource` interface requires that the resource managers be able to support the two-phase commit protocol from any thread context.

Transaction association

Transactions are associated with a transactional resource via the `start` method, and disassociated from the resource via the `end` method. The resource adapter is responsible for internally maintaining an association between the resource connection object and the `XAResource` object. At any given time, a connection is associated with a single transaction, or it is not associated with any transaction at all. Because JTA does not support nested

transactions it is an error for the `start` method to be invoked on a connection that is currently associated with a different transaction.

Interleaving multiple transaction contexts using the same resource may be done by the transaction manager as long as `start` and `end` are invoked properly for each transaction context switch. Each time the resource is used with a different transaction, the method `end` must be invoked for the previous transaction that was associated with the resource, and `start` must be invoked for the current transaction context.

Externally controlled connections

For transactional application whose transaction states are managed by an application server, its resources must also be managed by the application server so that transaction association is performed properly. If an application is associated with a transaction, it is an error for the application to perform transactional work through the connection without having the connection's resource object already associated with the global transaction. The application server must ensure that the `XAResource` object in use is associated with the transaction. This is done by invoking the `Transaction.enlistResource` method.

If a server side transactional application retains its database connection across multiple client requests, the application server must ensure that before dispatching a client request to the application thread, the resource is enlisted with the application's current transaction context. This implies that the application server manages the connection resource usage status across multiple method invocations.

Resource sharing

When the same transactional resource is used to interleave multiple transactions, it is the responsibility of the application server to ensure that only one transaction is enlisted with the resource at any given time. To initiate the transaction commit process, the transaction manager is allowed to use any of the resource objects connected to the same resource manager instance. The resource object used for the two-phase commit protocol does not need to have been involved with the transaction being completed.

The resource adapter must be able to handle multiple threads invoking the `XAResource` methods concurrently for transaction commit processing. For example, with reference to the code below, suppose we have a transactional resource `r1`. Global transaction `xid1` was started and ended with r1. Then a different global transaction xid2 is associated with r1. In the meanwhile, the transaction manager may start the two phase commit process for xid1 using r1 or any other transactional resource connected to the same resource manager. The resource adapter needs to allow the commit process to be executed while the resource is currently associated with a different global transaction.

```
XAResource xares = r1.getXAResource();

xares.start(xid1); // associate xid1 to the connection

..
xares.end(xid1); // disassociate xid1 to the connection
..
```

```
xares.start(xid2); // associate xid2 to the connection
..
// While the connection is associated with xid2,
// the TM starts the commit process for xid1
status = xares.prepare(xid1);
..
xares.commit(xid1, false);
```

Local and global transactions

The resource adapter must support the usage of both local and global transactions within the same transactional connection. Local transactions are transactions that are started and coordinated by the resource manager internally. The XAResource interface is not used for local transactions. When using the same connection to perform both local and global transactions, the following rules apply:

- The local transaction must be committed (or rolled back) before starting a global transaction in the connection.
- The global transaction must be disassociated from the connection before any local transaction is started.

Transaction timeouts

Timeout values can be associated with transactions in order to control their lifetime. If a transaction has not terminated (committed or rolled back) before the timeout value elapses, the transaction system will automatically roll it back. The `XAResource` interface supports a operation, which allows the timeout associated with the current transaction to be propagated to the resource manager and if supported, will override any default timeout associated with the resource manager. This can be useful when long running transactions may have lifetimes that would exceed the default and in which case, if the timeout were not altered, the resource manager would rollback before the transaction terminated and subsequently cause the transaction to roll back as well.

If no timeout value is explicitly set for a transaction, or a value of 0 is specified, then an implementation specific default value may be used. In the case of JBossTS use the CoordinatorEnvironmentBean.`defaultTimeout` property value and give a timeout in seconds. The default value is 60 seconds. If you set this to 0 then no timeout will be associated with transactions.

Unfortunately there are situations where imposing the same timeout as the transaction on a resource manager may not be appropriate. For example, if the system administrator wishes to have control over the lifetimes on resource managers and does not want to (or cannot) allow that control to be passed to some external entity. At present JBossTS supports an all-or-nothing approach to whether or not `setTransactionTimeout` is called on `XAResource` instances.

If the `JTAEnvironmentBean.xaTransactionTimeoutEnabled` property is set to `true` (the default) then it will be called on all instances. Alternatively, the

`setXATransactionTimeoutEnabled` method of `com.arjuna.ats.jta.common.Configuration` can be used.

Dynamic Registration

Dynamic registration is not supported in `XAResource` because of the following reasons:

- In the Java component-based application server environment, connections to the resource manager are acquired dynamically when the application explicitly requests a connection. These resources are enlisted with the transaction manager on a needed basis.
- If a resource manager requires a way to dynamically register its work to the global transaction, the implementation can be done at the resource adapter level via a private interface between the resource adapter and the underlying resource manager.

Chapter 1

Transaction recovery

Failure recovery

During recovery, the Transaction Manager needs to be able to communicate to all resource managers that are in use by the applications in the system. For each resource manager, the Transaction Manager uses the `XAResource.recover` method to retrieve the list of transactions that are currently in a prepared or heuristically completed state. Typically, the system administrator configures all transactional resource factories that are used by the applications deployed on the system. An example of such a resource factory is the JDBC `XADataSource` object, which is a factory for the JDBC `XAConnection` objects.

Because `XAResource` objects are not persistent across system failures, the Transaction Manager needs to have some way to acquire the `XAResource` objects that represent the resource managers which might have participated in the transactions prior to the system failure. For example, a Transaction Manager might, through the use of JNDI lookup mechanism, acquire a connection from each of the transactional resource factories, and then obtain the corresponding `XAResource` object for each connection. The Transaction Manager then invokes the `XAResource.recover` method to ask each resource manager to return the transactions that are currently in a prepared or heuristically completed state.

Note: When running XA recovery it is necessary to tell JBossTS which types of Xid it can recover. Each Xid that JBossTS creates has a unique node identifier encoded within it and JBossTS will only recover transactions and states that match a specified node identifier. The node identifier to use should be provided to JBossTS via the property `JTAEnvironmentBean.xaRecoveryNodes`; multiple values may be provided in a list. A value of '*' will force JBossTS to recover (and possibly rollback) all transactions irrespective of their node identifier and should be used with caution.

If using the *JBossJTA* JDBC driver, then *JBossJTA* will take care of all `XAResource` crash recovery automatically. Otherwise one of the following recovery mechanisms will be used:

- If the `XAResource` is serializable, then the serialized form will be saved during transaction commitment, and used during recovery. It is assumed that the recreated `XAResource` is valid and can be used to drive recovery on the associated database.
- The `com.arjuna.ats.jta.recovery.XAResourceRecovery`, `com.arjuna.ats.jta.recovery.XARecoveryResourceManager` and `com.arjuna.ats.jta.recovery.XARecoveryResource` interfaces are used. These are documented in the *JDBC chapters* on failure recovery.

Recovering XAConnections

When recovering from failures, *JBossJTA* requires the ability to reconnect to databases that were in use prior to the failures in order to resolve any outstanding transactions. Most connection information will be saved by the transaction service during its normal execution, and can be used during recovery to recreate the connection. However, it is possible that not all such information will have been saved prior to a failure (for example, a failure occurs before such information can be saved, but after the database connection is used). In order to recreate those connections it is necessary to provide implementations of the following *JBossJTA* interface `com.arjuna.ats.jta.recovery.XAResourceRecovery`, one for each database that may be used by an application.

Note: if using the transactional JDBC driver provided with JBossJTA, then no additional work is necessary in order to ensure that recovery occurs.

To inform the recovery system about each of the `XAResourceRecovery` instances, it is necessary to specify their class names through the JTAEnvironmentBean.xaResourceRecoveryInstances property variable, whose values is a list of space separated strings, each being a classname followed by optional configuration information.

```
JTAEnvironmentBean.xaResourceRecoveryInstances=com.foo.barRecovery
```

Additional information that will be passed to the instance when it is created may be specified after a semicolon:

```
JTAEnvironmentBean.xaResourceRecoveryInstances=com.foo.barRecovery;myData=h
ello
```

Note: These properties need to go into the JTA section of the property file.

Any errors will be reported during recovery.

```
public interface XAResourceRecovery
{
    public XAResource getXAResource () throws SQLException;

    public boolean initialise (String p);

    public boolean hasMoreResources ();
};
```

Each method should return the following information:

- *initialise*: once the instance has been created, any additional information which occurred on the property value (anything found after the first semi-colon) will be passed to the object. The object can then use this information in an implementation specific manner to initialise itself, for example.
- *hasMoreResources*: each `XAResourceRecovery` implementation may provide multiple `XAResource` instances. Before any call to `getXAResource` is made, `hasMoreResources` is called to determine whether there are any further

connections to be obtained. If this returns *false*, `getXAResource` will not be called again during this recovery sweep and the instance will not be used further until the next recovery scan. It is up to the implementation to maintain the internal state backing this method and to reset the iteration as required. Failure to do so will mean that the second and subsequent recovery sweeps in the lifetime of the JVM do not attempt recovery.

- *getXAResource*: returns an instance of the `XAResource` object. How this is created (and how the parameters to its constructors are obtained) is up to the `XAResourceRecovery` implementation. The parameters to the constructors of this class should be similar to those used when creating the initial driver or data source, and should obviously be sufficient to create new `XAResources` that can be used to drive recovery.

Note: If you want your XAResourceRecovery instance to be called during each sweep of the recovery manager then you should ensure that once hasMoreResources returns *false* to indicate the end of work for the current scan it then returns *true* for the next recovery scan.

Alternative to XAResourceRecovery

The iterator based approach used by XAResourceRecovery leads to a requirement for implementations to manage state, which makes them more complex than necessary.

As an alternative, starting with JBossTS 4.4, users may provide an implementation of the public interface

```
com.arjuna.ats.jta.recovery.XAResourceRecoveryHelper
{
   public boolean initialise(String p) throws Exception;
   public XAResource[] getXAResources() throws Exception;
}
```

During each recovery sweep the getXAResources method will be called and recovery attempted on each element of the array. For the majority of resource managers it will be necessary to have only one XAResource in the array, as the recover() call on it can return multiple Xids.

Unlike XAResourceRecovery instances, which are configured via the xml properties file and instantiated by JBossTS, instances of XAResourceRecoveryHelper and constructed by the application code and registered with JBossTS by calling

XARecoveryModule.addXAResourceRecoveryHelper(...)

The initialize method is not called by JBossTS in the current implementation, but is provided to allow for the addition of further configuration options in later releases.

XAResourceRecoveryHelper instances may be deregistered, after which they will no longer be called by the recovery manager. Deregistration may block for a time if a recovery scan is in progress.

XARecoveryModule.removeXAResourceRecoveryHelper(...)

The ability to dynamically add and remove instances of XAResourceRecoveryHelper whilst the system is running makes this approach an attractive option for environments in which e.g. datasources may be deployed or undeployed, such as application servers. Care should be taken with classloading behaviour in such cases.

Shipped XAResourceRecovery implementations

Recovery of XA datasources can sometimes be implementation dependant, requiring developers to provide their own XAResourceRecovery instances. However, JBossTS ships with several out-of-the-box implementations that may be useful.

Note: These XAResourceRecovery instances are primarily intended for when running JBossTS outside of a container such as JBossAS, since they rely upon XADataSources as the primary handle to drive recovery. If you are not running JBossTS stand-alone then you should consult the relevant integration documentation to ensure that the right recovery modules are being used.

- `com.arjuna.ats.internal.jdbc.recovery.BasicXARecovery`: this expects an XML property file to be specified upon creation and from which it will read the configuration properties for the datasource. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"><
<properties>
  <entry key="DB_X_DatabaseUser">username</entry>
  <entry key="DB_X_DatabasePassword">password"</entry>
  <entry key="DB_X_DatabaseDynamicClass">DynamicClass</entry>
  <entry key="DB_X_DatabaseURL">theURL</entry>
</properties>
```

- `com.arjuna.ats.internal.jdbc.recovery.JDBCXARecovery`: this recovery implementation should work on any datasource that is exposed via JNDI. It expects an XML property file to be specified upon creation and from which it will read the database JNDI name, username and password. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"><
<properties>
      <entry key="DatabaseJNDIName">java:ExampleDS</entry>
      <entry key="UserName">username</entry>
      <entry key="Password">password</entry>
</properties>
```

Because these classes are `XAResourceRecovery` instances they are passed any necessary initialization information via the `initialise` operation. In the case of BasicXARecovery and JDBCXARecovery this should be the location of a property file and is specified in the JBossTS configuration file. For example:

```
com.arjuna.ats.jta.recovery.XAResourceRecoveryJDBC=com.arjuna.ats.internal.
jdbc.recovery.JDBCXAResourceRecovery;thePropertyFile
```

Chapter 1

JDBC and transactions

Using the transactional JDBC driver

JBossJTA supports the construction of both local and distributed transactional applications which access databases using the JDBC APIs. JDBC supports two-phase commit of transactions, and is similar to the XA X/Open standard. The JDBC support is found in the `com.arjuna.ats.jdbc` package. A list of the tested drivers is available from the JBossTS website.

Note: The transactional JDBC support provided by JBossTS is intended for use when using JBossTS outside of containers such as JBossAS. You should not use them when running JBossTS in an embedded mode without first consulting your deployment documentation.

Managing transactions

JBossJTA must be able to associate work performed on a JDBC connection with a specific transaction. Therefore, implicit transaction propagation and/or indirect transaction management must be used by applications, i.e., for each JDBC connection it must be possible for *JBossJTA* to determine the invoking thread's current transaction context.

Restrictions

The following restrictions are imposed by limitations in the JDBC specifications and by *JBossJTA* to ensure that transactional interactions with JDBC databases can be correctly managed:

Nested transactions are not supported by JDBC If an attempt is made to use a JDBC connection within a subtransaction, *JBossJTA* will throw a suitable exception and no work will be allowed on that connection. However, if you wish to have nested transactions, then you can set the `com.arjuna.ats.jta.supportSubtransactions` property to `YES`.

Transactional drivers

The *JBossJTA* approach to incorporating JDBC connections within transactions is to provide transactional JDBC drivers through which all interactions occur. These drivers intercept all invocations and ensure that they are registered with, and driven by, appropriate transactions. There is a single type of transactional driver through which any JDBC driver can be driven; obviously if the database is not transactional then ACID properties cannot be guaranteed. This driver is `com.arjuna.ats.jdbc.TransactionalDriver`, which implements the `java.sql.Driver` interface.

Loading drivers

The driver may be directly instantiated and used within an application. For example:

```
TransactionalDriver arjunaJDBC2Driver = new TransactionalDriver();
```

They can be registered with the JDBC driver manager (`java.sql.DriverManager`) by adding them to the Java system properties. The `jdbc.drivers` property contains a list of driver class names, separated by colons, that are loaded by the JDBC driver manager when it is initialized.

```
/*
 * Register the driver via the system properties variable
 * "jdbc.drivers"
 */

Properties p = System.getProperties();

switch (dbType)
{
case MYSQL:
    p.put("jdbc.drivers", "com.mysql.jdbc.Driver");
    break;
case PGSQL:
    p.put("jdbc.drivers", "org.postgresql.Driver");
    break;
}

System.setProperties(p);
```

Alternatively, the `Class.forName()` method may be used to load the driver or drivers:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Calling `Class.forName()` will automatically register the driver with the JDBC driver manager. It is also possible to explicitly create an instance of the JDBC driver:

```
sun.jdbc.odbc.JdbcOdbcDriver drv = new sun.jdbc.odbc.JdbcOdbcDriver();

DriverManager.registerDriver(drv);
```

When you have loaded a driver, it is available for making a connection with a DBMS.

Connections

In this section we shall discuss the notion of transactional JDBC connections, how they are managed within *JBossJTA* and the implications on using them within an application.

Making the connection

Because JDBC connectivity in *JBossJTA* works by simply providing a new JDBC driver, application code can remain relatively the same to that when not using transactions. Typically, the application programmer need only start and terminate transactions.

JDBC

Before describing the JDBC support it is necessary to mention that the following properties can be set and passed to the *JBossJTA* driver (they are all located in the `com.arjuna.ats.jdbc.TransactionalDriver` class):

- *userName*: the user name to use when attempting to connect to the database.
- *password*: the password to use when attempting to connect to the database.
- *createDb*: if set to true, the driver will attempt to create the database when it connects. This may not be supported by all JDBC implementations.
- *dynamicClass*: this specifies a class to instantiate to connect to the database, rather than using JNDI.

XADataSources

JDBC connections are created from appropriate `DataSources`. Those connections which must participate within distributed transactions are obtained from `XADataSources`. Therefore, when using a JDBC driver, *JBossJTA* will use the appropriate `DataSource` whenever a connection to the database is made. It will then obtain `XAResources` and register them with the transaction via the JTA interfaces. It is these `XAResources` which the transaction service will use when the transaction terminates in order to drive the database to either commit or rollback the changes made via the JDBC connection.

There are two ways in which the *JBossJTA* JDBC support can obtain `XADataSources`. These will be explained in the following sections. Note, for simplicity we shall assume that the JDBC driver is instantiated directly by the application.

Java Naming and Directory Interface (JNDI)

In order to allow a JDBC driver to use arbitrary `DataSources` without having to know specific details about their implementations, `DataSources` are typically obtained from JNDI. A specific `(XA)DataSource` can be created and registered with an appropriate JNDI implementation, and the application (or JDBC driver) can later bind to and use it. Since JNDI only allows the application to see the `(XA)DataSource` as an instance of the interface (e.g., `javax.sql.XADataSource`) rather than as an instance of the implementation class (e.g., `com.mydb.myXADataSource`), the application is not tied at build time to only use a specific `(XA)DataSource` implementation.

To get the `TransactionalDriver` class to use a JNDI registered `XADataSource` it is first necessary to create the `XADataSource` instance and store it in an appropriate JNDI implementation. Details of how to do this can be found in the JDBC tutorial available at the Java web site. An example is show below:

```
XADataSource ds = MyXADataSource();
Hashtable env = new Hashtable();
String initialCtx =
PropertyManager.getProperty("Context.INITIAL_CONTEXT_FACTORY");
```

```
env.put(Context.INITIAL_CONTEXT_FACTORY, initialCtx);

initialContext ctx = new InitialContext(env);

ctx.bind("jdbc/foo", ds);
```

Where the `Context.INITIAL_CONTEXT_FACTORY` property is the JNDI way of specifying the type of JNDI implementation to use.

Then the application must pass an appropriate connection URL to the JDBC driver:

```
Properties dbProps = new Properties();

dbProps.setProperty(TransactionalDriver.userName, "user");
dbProps.setProperty(TransactionalDriver.password, "password");

// the driver uses its own JNDI context info, remember to set it up:
jdbcPropertyManager.propertyManager.setProperty(
  "Context.INITIAL_CONTEXT_FACTORY", initialCtx);
jdbcPropertyManager.propertyManager.setProperty(
  "Context.PROVIDER_URL", myUrl);

TransactionalDriver arjunaJDBCDriver = new TransactionalDriver();
Connection connection = arjunaJDBCDriver.connect("jdbc:arjuna:jdbc/foo",
dbProps);
```

The JNDI URL must be pre-pended with *jdbc:arjuna:* in order for the `TransactionalDriver` to recognise that the `DataSource` must participate within transactions and be driven accordingly.

Dynamic class instantiation

In some cases a JNDI implementation is not available. For such situations, it is possible to specify an implementation of the `DynamicClass` interface, which will be used to get the `XADataSource` object. **This is not recommended**, but provides a fallback for environments where use of JNDI is not feasible.

Use the property `TransactionalDriver.dynamicClass` to specify the implementation to use. As an example, we provide `PropertyFileDynamicClass`, a `DynamicClass` implementation that reads the `XADataSource` implementation class name and configuration properties from a file, then instantiates and configures it.

Note: the oracle_8_1_6 dynamic class is deprecated and should not be used.

The application code must specify which dynamic class the `TransactionalDriver` should instantiate when setting up the connection:

```
Properties dbProps = new Properties();

dbProps.setProperty(TransactionalDriver.userName, "user");
dbProps.setProperty(TransactionalDriver.password, "password");
dbProps.setProperty(TransactionalDriver.dynamicClass,
                                        "com.arjuna.ats.internal.jdbc.d
rivers.PropertyFileDynamicClass);
```

```
TransactionalDriver arjunaJDBC2Driver = new TransactionalDriver();
Connection connection =
arjunaJDBC2Driver.connect("jdbc:arjuna:/path/to/property/file",
dbProperties);
```

Using the connection

Once the connection has been established (for example, using the `java.sql.DriverManager.getConnection` method), all operations on the connection will be monitored by *JBossJTA*. Note, it is not necessary to use the transactional connection within transactions. If a transaction is not present when the connection is used, then operations will be performed directly on the database.

Note: JDBC does not support subtransactions.

Transaction timeouts can be used to automatically terminate transactions should the connection not be terminated within an appropriate period.

JBossJTA connections can be used within multiple different transactions simultaneously, i.e., different threads, with different notions of the current transaction, may use the same JDBC connection. *JBossJTA* does connection pooling for each transaction within the JDBC connection. So, although multiple threads may use the same instance of the JDBC connection, internally this may be using a different connection instance per transaction. With the exception of *close*, all operations performed on the connection at the application level will only be performed on this transaction-specific connection.

JBossJTA will automatically register the JDBC driver connection with the transaction via an appropriate resource. When the transaction terminates, this resource will be responsible for either committing or rolling back any changes made to the underlying database via appropriate calls on the JDBC driver.

Once created, the driver and any connection can be used in the same way as any other JDBC driver or connection.

```
Statement stmt = conn.createStatement();

try
{
    stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b INTEGER)");
}
catch (SQLException e)
{
  // table already exists
}

stmt.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");

ResultSet res1 = stmt.executeQuery("SELECT * FROM test_table");
```

Connection pooling

For each user name and password, *JBossJTA* will maintain a single instance of each connection for as long as that connection is in use. Subsequent requests for the same connection will get a reference to the originally created connection, rather than a new instance. Attempts to close the connection are allowed, but the connection will only actually be closed when all users (including transactions) have either finished with the connection, or issued close calls.

Reusing connections

Some JDBC drivers allow the reuse of a connection for multiple different transactions once a given transaction has completed. Unfortunately this is not a common feature, and other drivers require a new connection to be obtained for each new transaction. By default, the *JBossJTA* transactional driver will always obtain a new connection for each new transaction. However, if an existing connection is available and is currently unused, it is possible to make *JBossJTA* reuse this connection. In order to do this, the `reuseconnection=true` option must be specified on the JDBC URL. For example:

```
jdbc:arjuna:sequelink://host:port;databaseName=foo;reuseconnection=true
```

Terminating the transaction

Whenever a transaction terminates (either explicitly by the application programmer, or implicitly when any associated transaction timeout expires) that has a JDBC connection registered with it, *JBossJTA* will drive the database (via the JDBC driver) to either commit or roll back any changes made to it. This happens transparently to the application.

AutoCommit

If `AutoCommit` of the `java.sql.Connection` is set to `true` for JDBC then the execution of every SQL statement is a separate top-level transaction, and grouping multiple statements to be managed within a single OTS transaction is not possible. Therefore, *JBossJTA* will disable `AutoCommit` on JDBC connections before they can be used. If auto commit is subsequently set to true by the application, *JBossJTA* will raise the `java.sql.SQLException`.

Setting isolation levels

When using the *JBossJTA* JDBC driver, it may be necessary to set the underlying transaction isolation level on the XA connection. By default, this is set to `TRANSACTION_SERIALIZABLE`, but you may want to set this to something more appropriate for your application. In order to do this, set the `com.arjuna.ats.jdbc.isolationLevel` property to the appropriate isolation level in string form, e.g., `TRANSACTION_READ_COMMITTED`, or `TRANSACTION_REPEATABLE_READ`.

Note: At present this property applies to all XA connections created in the JVM.

Chapter 1

Examples

JDBC example

The following code illustrates many of the points described above (note that for simplicity, much error checking code has been remove). This example assumes that you are using the transactional JDBC driver provided with JBossTS. For details about how to configure and use this driver see the previous Chapter.

```
public class JDBCTest
{
public static void main (String[] args)
{
    /*
     */

    Connection conn = null;
    Connection conn2 = null;
    Statement stmt = null;        // non-tx statement
    Statement stmtx = null;      // will be a tx-statement
    Properties dbProperties = new Properties();

    try
    {
        System.out.println("\nCreating connection to database: "+url);

        /*
         * Create conn and conn2 so that they are bound to the JBossTS
         * transactional JDBC driver. The details of how to do this will
         * depend on your environment, the database you wish to use and
         * whether or not you want to use the Direct or JNDI approach. See
         * the appropriate chapter in the JTA Programmers Guide.
         */

        stmt = conn.createStatement();  // non-tx statement

        try
        {
          stmt.executeUpdate("DROP TABLE test_table");
          stmt.executeUpdate("DROP TABLE test_table2");
        }
        catch (Exception e)
        {
            // assume not in database.
        }

        try
        {
            stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b
INTEGER)");
            stmt.executeUpdate("CREATE TABLE test_table2 (a INTEGER,b
INTEGER)");
        }
        catch (Exception e)
        {
```

```
        }

        try
        {
            System.out.println("Starting top-level transaction.");

            com.arjuna.ats.jta.UserTransaction.userTransaction().begin();

            stmtx = conn.createStatement(); // will be a tx-statement

            System.out.println("\nAdding entries to table 1.");

            stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES
(1,2)");

            ResultSet res1 = null;

            System.out.println("\nInspecting table 1.");

            res1 = stmtx.executeQuery("SELECT * FROM test_table");
            while (res1.next())
            {
                System.out.println("Column 1: "+res1.getInt(1));
                System.out.println("Column 2: "+res1.getInt(2));
            }

            System.out.println("\nAdding entries to table 2.");

            stmtx.executeUpdate("INSERT INTO test_table2 (a, b) VALUES
(3,4)");
            res1 = stmtx.executeQuery("SELECT * FROM test_table2");
            System.out.println("\nInspecting table 2.");

            while (res1.next())
            {
                System.out.println("Column 1: "+res1.getInt(1));
                System.out.println("Column 2: "+res1.getInt(2));
            }
            System.out.print("\nNow attempting to rollback changes.");
            com.arjuna.ats.jta.UserTransaction.userTransaction().rollback()
;

            com.arjuna.ats.jta.UserTransaction.userTransaction().begin();
            stmtx = conn.createStatement();
            ResultSet res2 = null;

            System.out.println("\nNow checking state of table 1.");

            res2 = stmtx.executeQuery("SELECT * FROM test_table");
            while (res2.next())
            {
                System.out.println("Column 1: "+res2.getInt(1));
                System.out.println("Column 2: "+res2.getInt(2));
            }

            System.out.println("\nNow checking state of table 2.");

            stmtx = conn.createStatement();
            res2 = stmtx.executeQuery("SELECT * FROM test_table2");
            while (res2.next())
            {
                System.out.println("Column 1: "+res2.getInt(1));
                System.out.println("Column 2: "+res2.getInt(2));
```

```
            }

            com.arjuna.ats.jta.UserTransaction.userTransaction().commit(tru
e);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            System.exit(0);
        }
    }
    catch (Exception sysEx)
    {
        sysEx.printStackTrace();
        System.exit(0);
    }
}
```

Failure recovery example with BasicXARecovery

This class implements the XAResourceRecovery interface for XAResources. The parameter supplied in setParameters can contain arbitrary information necessary to initialize the class once created. In this instance it contains the name of the property file in which the db connection information is specified, as well as the number of connections that this file contains information on (separated by ;).

> **Caution:** This is only an *example* of the sorts of things an XAResourceRecovery implementer could do. This implementation uses a property file that is assumed to contain sufficient information to recreate connections used during the normal run of an application so that we can perform recovery on them. It is not recommended that information such as user name and password appear in such a raw text format as it opens up a potential security hole.

The db parameters specified in the property file are assumed to be in the format:

- DB_x_DatabaseURL=
- DB_x_DatabaseUser=
- DB_x_DatabasePassword=
- DB_x_DatabaseDynamicClass=

Where x is the number of the connection information.

> **Note:** Some error handling code has been removed from this text for ease of readability purposes.

```
/*
 * Some XAResourceRecovery implementations will do their startup work here,
 * and then do little or nothing in setDetails. Since this one needs to know
 * dynamic class name, the constructor does nothing.
 */

public BasicXARecovery () throws SQLException
{
```

```
        numberOfConnections = 1;
        connectionIndex = 0;
        props = null;
}

/**
 * The recovery module will have chopped off this class name already. The
 * parameter should specify a property file from which the url, user name,
 * password, etc. can be read.
 *
 * @message com.arjuna.ats.internal.jdbc.recovery.initexp An exception
 *          occurred during initialisation.
 */

public boolean initialise (String parameter) throws SQLException
{
        if (parameter == null)
                return true;

        int breakPosition = parameter.indexOf(BREAKCHARACTER);
        String fileName = parameter;

        if (breakPosition != -1)
        {
                fileName = parameter.substring(0, breakPosition - 1);

                try
                {
                        numberOfConnections = Integer.parseInt(parameter
                                        .substring(breakPosition + 1));
                }
                catch (NumberFormatException e)
                {
                        return false;
                }
        }

        try
        {
                String uri = com.arjuna.common.util.FileLocator
                                .locateFile(fileName);
                jdbcPropertyManager.propertyManager.load(XMLFilePlugin.class
                                .getName(), uri);

                props = jdbcPropertyManager.propertyManager.getProperties();
        }
        catch (Exception e)
        {
                return false;
        }

        return true;
}

/**
 * @message com.arjuna.ats.internal.jdbc.recovery.xarec {0} could not find
 *          information for connection!
 */

public synchronized XAResource getXAResource () throws SQLException
{
        JDBC2RecoveryConnection conn = null;

        if (hasMoreResources())
        {
                connectionIndex++;

                conn = getStandardConnection();
```

```
                if (conn == null) conn = getJNDIConnection();
        }

        return conn.recoveryConnection().getConnection().getXAResource();
}

public synchronized boolean hasMoreResources ()
{
        if (connectionIndex == numberOfConnections)
                return false;
        else
                return true;
}

private final JDBC2RecoveryConnection getStandardConnection ()
                throws SQLException
{
        String number = new String("" + connectionIndex);
        String url = new String(dbTag + number + urlTag);
        String password = new String(dbTag + number + passwordTag);
        String user = new String(dbTag + number + userTag);
        String dynamicClass = new String(dbTag + number + dynamicClassTag);

        Properties dbProperties = new Properties();

        String theUser = props.getProperty(user);
        String thePassword = props.getProperty(password);

        if (theUser != null)
        {
                dbProperties.put(TransactionalDriver.userName, theUser);
                dbProperties.put(TransactionalDriver.password, thePassword);

                String dc = props.getProperty(dynamicClass);

                if (dc != null)
                        dbProperties.put(TransactionalDriver.dynamicClass, dc);

                return new JDBC2RecoveryConnection(url, dbProperties);
        }
        else
                return null;
}

private final JDBC2RecoveryConnection getJNDIConnection ()
                throws SQLException
{
        String number = new String("" + connectionIndex);
        String url = new String(dbTag + jndiTag + number + urlTag);
        String password = new String(dbTag + jndiTag + number + passwordTag);
        String user = new String(dbTag + jndiTag + number + userTag);

        Properties dbProperties = new Properties();

        String theUser = props.getProperty(user);
        String thePassword = props.getProperty(password);

        if (theUser != null)
        {
                dbProperties.put(TransactionalDriver.userName, theUser);
                dbProperties.put(TransactionalDriver.password, thePassword);

                return new JDBC2RecoveryConnection(url, dbProperties);
        }
        else
                return null;
}

private int numberOfConnections;
```

```
private int connectionIndex;
private Properties props;
private static final String dbTag = "DB_";
private static final String urlTag = "_DatabaseURL";
private static final String passwordTag = "_DatabasePassword";
private static final String userTag = "_DatabaseUser";
private static final String dynamicClassTag = "_DatabaseDynamicClass";
private static final String jndiTag = "JNDI_";

/*
 * Example:
 *
 * DB2_DatabaseURL=jdbc\:arjuna\:sequelink\://qa02\:20001
 * DB2_DatabaseUser=tester2 DB2_DatabasePassword=tester
 * DB2_DatabaseDynamicClass=com.arjuna.ats.internal.jdbc.drivers.sequelink_5_1
 *
 * DB_JNDI_DatabaseURL=jdbc\:arjuna\:jndi DB_JNDI_DatabaseUser=tester1
 * DB_JNDI_DatabasePassword=tester DB_JNDI_DatabaseName=empay
 * DB_JNDI_Host=qa02 DB_JNDI_Port=20000
 */

    private static final char BREAKCHARACTER = ';'; // delimiter for parameters
```

The class `com.arjuna.ats.internal.jdbc.recovery.JDBC2RecoveryConnection` may be used to create a new connection to the database using the same parameters that were used to create the initial connection.

Chapter 1

Configuring JBossJTA

Configuration options

The following table shows the configuration features, with default values shown in italics. For more detailed information, the relevant section numbers are provided.

Configuration Name	Possible Values	Relevant Section
com.arjuna.ats.jta.supportSubtransactions	*YES*/NO	
com.arjuna.ats.jta.jtaTMImplementation	com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple/com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple	
com.arjuna.ats.jta.jtaUTImplementation	com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple/com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImple	,
com.arjuna.ats.jta.xaBackoffPeriod		
com.arjuna.ats.jdbc.isolationLevel	Any supported JDBC isolation level.	
com.arjuna.ats.jta.xaTransactionTimeoutEnabled	*true*/false	Chapter 3

Table 2: JBossJTA configuration options.

Chapter 2

Using JBossJTA in application servers

JBOSS Application Server

Configuration

Service Configuration

The JBoss Transaction Service is configured primarily via the XML files stored in the etc directory, but when run as a JBOSS service there are a number of configurable attributes available. They are as follows:

TransactionTimeout – The default transaction timeout to be used for new transactions. Specified as an integer in seconds.

StatisticsEnabled – This determines whether or not the transaction service should gather statistical information. This information can then be viewed using the PerformanceStatistics MBean. Specified as a Boolean. The default is to not gather this information.

PropagateFullContext – This determines whether a full transactional context is propagated by context importer/exporter. If set to false only the current transaction context is propagated. If set to true the full transaction context (including parent transactions) is propagated.

These attributes are specified as MBean attributes in the `jboss-service.xml` file located in the `server/all/conf` directory, e.g.

```
<mbean  code="com.arjuna.ats.jbossatx.jts.TransactionManagerService"
name="jboss:service=TransactionManager">

<attribute name="TransactionTimeout">300</attribute>
<attribute name="StatisticsEnabled">true</attribute>

</mbean>
```

The transaction service is configurable also via the standard JBoss Transaction Service property files. These are located in the JBossTS install location under the `etc` sub-directory. These files can be edited manually or through JMX. Each property file is exposed via an object with the name `com.arjuna.ts.properties` and an attribute of `module` where module is equal to the name of the module to be configured, e.g. `com.arjuna.ts.properties:module=arjuna.`

Logging

In order to make JBossTS logging semantically consistent with JBossAS, the TransactionManagerService modifies the level of some log messages. This is achieved by overriding the value of the com.arjuna.common.util.logger property given in the jbossts-properties.xml file. Therefore, the value of this property will have no effect on the logging behaviour when running embedded in JBossAS. By forcing use of the log4j_releveler logger, the TransactionManagerService causes all INFO level messages in the transaction code to be modified to behave as DEBUG messages. Therefore, these messages will not appear in log files if the filter level is INFO. All other log messages behave as normal.

The services

There is currently one service offered by the JBOSS integration. In this section we shall discuss what this service does.

TransactionManagerService

The transaction manager service's main purpose is to ensure the recovery manager is started. It also binds the JBossTS JTA transaction manager to `java:/TransactionManager` name with the JNDI provider. This service depends upon the existence of the CORBA ORB Service and it must be using JacORB as the underlying ORB implementation.

There are two instances of this service:

- distributed: this uses the JTS enabled transaction manager implementation and hence supports distributed transactions and recovery. To configure this use the `com.arjuna.ats.jbossatx.jts.TransactionManagerService` class. This is the default configuration.
- local: this uses the purely local JTA implementation. To configure this use the `com.arjuna.ats.jbossatx.jta.TransactionManagerService` class.

Ensuring Transactional Context is Propagated to the Server

It is possible to coordinate transactions from a coordinator which is not located within the JBoss server (e.g. using transactions created by an external OTS server). To ensure the transaction context is propagated via JRMP invocations to the server, the transaction propagation context factory needs to be explicitly set for the JRMP invoker proxy. This is done as follows:

```
JRMPInvokerProxy.setTPCFactory( new
com.arjuna.ats.internal.jbossatx.jts.PropagationContextManager() );
```

Index