JBoss Enterprise Application Platform Common Criteria Certification 5

Transactions Development Guide

Developing Applications Using JTA, JTS, and XTS APIs

Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5 Transactions Development Guide

Developing Applications Using JTA, JTS, and XTS APIs
Edition 5.1.0

Mark Little
Red Hat
mlittle@redhat.com

Andrew Dinn
Red Hat
adinn@redhat.com

Kevin Connor
Red Hat
kconnor@redhat.com

Jonathan Halliday
Red Hat
jhallida@redhat.com

Edited by

Misty Stanley-Jones
Red Hat
misty@redhat.com

Legal Notice

Abstract

This book is intended to assist Java developers in creating transactional applications using the JBoss implementations of the JTA, JTS, and XTS APIs.

Table of Contents

PART I. JTA DEVELOPMENT

This section gives guidance for using the JBoss implementation of the Java Transactions (JTA) API to add transactional support for your enterprise applications.

CHAPTER 1. AN INTRODUCTION TO THE JAVA TRANSACTION API (JTA)

Transactional standards provide extremely low-level interfaces for use by application programmers. Sun Microsystems has specified higher-level interfaces to assist in the development of distributed transactional applications. These interfaces are still low-level enough to require the programmer to be concerned with state management and concurrency for transactional application. They are most useful 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

> Provides the infrastructure required to support an application run-time environment which includes transaction state management, such as an EJB server.

Transaction Manager

> Provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, 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. The transaction manager uses this interface to communicate transaction association, transaction completion, and recovery.

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. High-level interfaces allow transaction interface users to drive transaction demarcation, resource enlistment, synchronization, and recovery processes. 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 JTA 1.0.1 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. THE JBOSS JTA IMPLEMENTATION

The Java Transaction API (JTA) consists of three elements:

- A high-level application transaction demarcation interface

- A high-level transaction manager interface intended for application server

- A standard Java mapping of the X/Open XA protocol intended for transactional resource manager

All of the JTA classes and interfaces are declared within the javax.transaction package, and the corresponding JBossJTA implementations are defined within the com.arjuna.ats.jta package.

> **IMPORTANT**
>
> Each Xid that JBoss Transaction Service creates needs a unique node identifier encoded within it. JBoss Transaction Service will only recover transactions and states that match a specified node identifier. The node identifier should be provided to JBoss transaction Service via the com.arjuna.ats.arjuna.xa.nodeIdentifier property. You must ensure this value is unique across your JBoss Transaction Service instances. If you do not provide a value, JBoss Transaction Service will generate one and report the value via the logging infrastructure. The node identifier should be alphanumeric.

2.1. USERTRANSACTION

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

> **NOTE**
>
> You can obtain a **UserTransaction** from JNDI.
>
> ```
> InitialContext ic = new InitialContext();
> UserTransaction utx = ic.lookup("java:comp/UserTransaction")
> ```

In order to select the local JTA implementation:

1. Set the com.arjuna.ats.jta.jtaTMImplementation property to
   com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple.

2. Set the com.arjuna.ats.jta.jtaUTImplementation property to
   com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple.

2.2. TRANSACTIONMANAGER

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

> **NOTE**
>
> You can obtain a **TransactionManager** from JNDI.
>
> ```
> InitialContext ic = new InitialContext();
> TransactionManager utm = ic.lookup("java:/TransactionManager")
> ```

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 can be associated with the same global transaction. Nested transactions are not supported.

Each transaction context is encapsulated within 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** begins 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 the **begin** method 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. These operations are described elsewhere.

The **commit** method completes the transaction currently associated with the calling thread. After it returns, the calling thread is not associated with any transaction. If **commit** is called when the thread is not associated with any transaction context, an exception is thrown. In some implementations, only the transaction originator can use the **commit** operation. If the calling thread is not permitted to commit the transaction, an exception is thrown. JBossJTA does not impose any restrictions on the ability of threads to terminate transactions.

The **rollback** method is used to roll back the transaction associated with the current thread. After the **rollback** method completes, the thread is not associated with any transaction.

> **NOTE**
>
> In a multi-threaded environment, multiple threads may be active within the same transaction. If *checked transaction semantics* have been disabled, or the transaction times out, then a transaction can be terminated by a thread other than its creator. If this happens, the creator must be notified. JBoss Transaction Service does this notification during commit or rollback by throwing the **IllegalStateException** exception.

2.3. 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 associated with the calling thread. 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 valid, the transaction context is associated with the calling thread. Otherwise, the thread is not associated with any transaction.
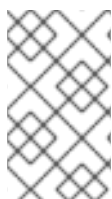
> **NOTE**
>
> If the **resume** method is invoked when the calling thread is already associated with another transaction, the Transaction Manager throws the **IllegalStateException** exception.

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

> **NOTE**
>
> JBossJTA supports allowing a suspended transaction to be resumed by a different thread, even though this feature is not required by the JTA standards.

When a transaction is suspended, the application server de-register and free up the resources that related to 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 give the transaction back its resources. 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.

2.4. 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. JBossJTA does not impose restrictions on threads terminating transactions.

2.5. 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 coordinate 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.

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 causes the transaction manager to inform 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 dissociate the specified resource from the transaction context in the target object. The application server invokes the method with two parameters:

- An **XAResources** object, which represents the resource.

- 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 causes the transaction manager to inform 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, in which case 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.

2.6. 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** callback object to be invoked by the transaction manager either before or after completion:

- 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 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.

2.7. 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 returns **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);
```

CHAPTER 3. THE RESOURCE MANAGER

3.1. THE XARESOURCE INTERFACE

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, and defines the contract between a *Resource Manager* and a *Transaction Manager* in a distributed transaction processing environment. A *resource adapter* implements the **XAResource** interface to support association of a top-level transaction to a resource. A relational database is an example of such a resource.

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. An application can 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. The **start** and **end** methods associates and dissociate the transaction from the resource.

The resource manager associates the transaction with all work performed on its data between the **start** and **end** invocations. At transaction commit time, these transactional resource managers are instructed 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 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 caused by the improper handling of the **XAResource** object by the transaction manager are mapped to Java exceptions by the **XAException** class.

- The DTP concept of *Thread of Control* maps to all Java threads with access to the **XAResource** and **Connection** objects. For example, two different threads are able to perform the **start** and **end** operations on the same **XAResource** object.

3.1.1. 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. However, JBoss Transaction Service supports controlling the order with 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 at the beginning or end of the commit protocol.

NOTE

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

Last Resource Commit optimization (LRCO) allows a single resource that is only one-phase aware (does not support **prepare**) to be enlisted with a transaction which manipulates two-phase aware participants. JBossJTA provides LRCO support.

In order to use the LRCO feature, your **XAResource** implementation must extend the **com.arjuna.ats.jta.resources.LastResourceCommitOptimisation** marker interface. When enlisting the resource via **Transaction.enlistResource**, JBoss Transaction Service allows only a single **LastResourceCommitOptimisation** participant to be used within each transaction. Your resource is driven last in the commit protocol, and the **prepare** method is not invoked.

NOTE

By default, an attempt to enlist more than one instance of a **LastResourceCommitOptimisation** class will fail and **false** is returned from **Transaction.enlistResource**. You can override this behavior by setting the com.arjuna.ats.jta.allowMultipleLastResources property to **true**. Be sure to read the section on enlisting multiple one-phase aware resources fore more information.

To use the LRCO in a distributed environment, you must disable interposition support. You are still able to use implicit context propagation.

3.1.2. Enlisting Multiple One-Phase Aware Resources

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. When possessing a single one-phase aware resource, you can still achieve an atomic (all or nothing) outcome across resources by utilizing LRCO, as explained earlier.

However, you may have enlisted multiple one-phase aware resources within the same transaction. For example, a legacy database running within the same transaction as a legacy JMS implementation. In these situations, you cannot achieve atomicity of transaction outcome across multiple resources, because none of them enter the **prepare** state. They commit or rollback immediately when instructed by the transaction coordinator, without knowledge of other resource states and without any way of undoing their actions if subsequent resources make a different choice. This can cause data corruption or heuristic outcomes.

In these situations, use either of the following approaches:

- Wrap the resources in compensating transactions.

- Migrate the legacy implementations to two-phase aware equivalents.

If neither of these options are viable, JBoss Transaction Service supports the enlistment of multiple one-phase aware resources within the same transaction, using LRCO. LRCO is covered earlier in this chapter.
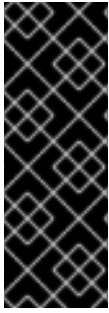
IMPORTANT

Even when LRCO support is enabled, JBoss Transaction Service issues warnings when it detects this support. The log message is **"You have chosen to enable multiple last resources in the transaction manager. This is transactionally unsafe and should not be relied upon."** or, when multiple one-phase resources are enlisted within the transaction, **"This is transactionally unsafe and should not be relied on."**.

3.2. OPENING A RESOURCE MANAGER

The X/Open **XA** interface requires the transaction manager to initialize a resource manager using the **xa_open** prior to issuing any other **xa_** calls. JTA requires initialization of a resource manager to be embedded within the resource adapter representing the resource manager. The transaction manager does not need to know how to initialize a resource manager. It must only tell the resource manager 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.

3.3. 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 the 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 explicitly released (closed). When the application invokes the connection's **close** method, the resource adapter invalidates the connection object reference that was held by the application, notifying the application server about the **close**. The transaction manager needs to invoke the **XAResource.end** method to dissociate the transaction from that connection.

The **close** notification allows the application server to perform any necessary garbage collection and mark the physical XA connection as free for reuse, in the case of connection pooling.

3.4. THREADS OF CONTROL

The X/Open **XA** interface specifies that the XA calls related to transaction associations 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 use the same connection resource to access the resource manager if the connection spans multiple method invocations. 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 operate independent of thread context. Therefore, different threads may invoke the **start** and **end** methods.

If the application server allows multiple threads to use a single **XAResource** object and its associated connection to the resource manager, the application server must ensure that only one transaction context is associated with the resource at any point in time. Therefore, the **XAResource** interface

requires the resource managers to be able to support the two-phase commit protocol from any thread context.

3.5. TRANSACTION ASSOCIATION

Transactions are associated with a transactional resource via the `start` method, and dissociated from the resource via the **end** method. The resource adapter internally maintains an association between the resource connection object and the **XAResource** object. At any given time, a connection is associated with zero or one transactions. Because JTA does not support nested transactions, the `start` method cannot be invoked on a connection that is currently associated with a different transaction.

The transaction manager may interleave multiple transaction contexts with the same resource, 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 **end** method must be invoked for the previous transaction that was associated with the resource, and the `start` method must be invoked for the current transaction context.

3.6. EXTERNALLY-CONTROLLED CONNECTIONS

If a transactional application's 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 incorrect 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 associate the **XAResource** object in use with the transaction by invoking the **Transaction.enlistResource** method.

If a server-side transactional application retains its database connection across multiple client requests, the application server must enlist the resource with the application's current transaction context. In this way, the application server manages the connection resource usage status across multiple method invocations.

3.7. RESOURCE SHARING

When the same transactional resource is used to interleave multiple transactions, the application server is responsible for ensuring that only one transaction is enlisted with the resource at any given time. To initiate the transaction `commit` process, the transaction manager can 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 be associated with the transaction being completed.

The resource adapter must be able to handle multiple threads invoking the **XAResource** methods concurrently for transaction `commit` processing. The code below declares a transactional resource `r1`. Global transaction `xid1` is started and ended with `r1`. Then a different global transaction `xid2` is associated with **r1**. In the meantime, 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);
```

3.8. 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 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 dissociated from the connection before any local transaction is started.

3.9. TRANSACTION TIMEOUTS

Timeout values can be associated with transactions for life cycle control. If a transaction has not terminated (committed or rolled back) before the timeout value elapses, the transaction system automatically rolls it back. The **XAResource** interface supports a operation allowing the timeout associated with the current transaction to be propagated to the resource manager and, if supported, overrides any default timeout associated with the resource manager. This is useful when long-running transactions have lifetimes that exceed the default. If the timeout is not altered, the resource manager will rollback before the transaction terminates 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 JBoss Transaction Service, how this default value is set depends upon which JTA implementation you are using.

Local JTA

Set the com.arjuna.ats.arjuna.coordinator.defaultTimeout property to a value expressed in seconds. The default value is 60 seconds.

JTS

Set the com.arjuna.ats.jts.defaultTimeout property to a value expressed in seconds. The default value is 0, meaning that transactions do not time out.

Unfortunately there are situations where imposing the same timeout as the transaction on a resource manager may not be appropriate. For example, the system administrator may need control over the lifetimes of resource managers without allowing that control to be passed to some external entity. JBoss Transaction Service supports an all-or-nothing approach to whether **setTransactionTimeout** is called on **XAResource** instances.

If the com.arjuna.ats.jta.xaTransactionTimeoutEnabled property is set to **true** (the default), it is called on all instances. Alternatively, the **setXATransactionTimeoutEnabled** method of **com.arjuna.ats.jta.common.Configuration** can be used.

3.10. DYNAMIC REGISTRATION

Dynamic registration is not supported in **XAResource** for 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 an as-needed basis.

- If a resource manager needs to dynamically register its work to the global transaction, it can be done at the resource adapter level via a private interface between the resource adapter and the underlying resource manager.

CHAPTER 4. TRANSACTION RECOVERY

4.1. FAILURE RECOVERY

During recovery, the Transaction Manager needs the ability 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 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. The JDBC **XADataSource** object, for example, is a factory for the JDBC **XAConnection** objects.

Because **XAResource** objects are not persistent across system failures, the Transaction Manager needs the ability to acquire the **XAResource** objects that represent the resource managers which might have participated in the transactions prior to a system failure. For example, a Transaction Manager might use the JNDI look-up mechanism to 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.
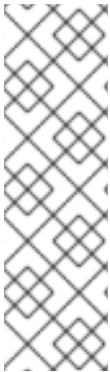
> **NOTE**
>
> When running XA recovery, you must tell JBoss Transaction Service which types of Xid it can recover. Each Xid that JBoss Transaction Service creates has a unique node identifier encoded within it, and JBoss Transaction Service only recovers transactions and states that match the requested node identifier. The node identifier to use should be provided to JBoss Transaction Service in a property that starts with the name com.arjuna.ats.jta.xaRecoveryNode. Multiple values are allowed. A value of * forces recovery, and possibly rollback, of all transactions, regardless of their node identifier. Use it with caution.

If the JBossJTA JDBC 2.0 driver is in use, JBossJTA manages all **XAResource** crash recovery automatically. Otherwise one, of the following recovery mechanisms is used:

- If the **XAResource** is able to be serialized, then the serialized form will be saved during transaction commitment, and used during recovery. The recreated **XAResource** is assumed to be valid and able 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. Refer to the JDBC chapters on failure recovery for more information.

4.2. 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 outstanding transactions. Most connection information is saved by the transaction service during its normal execution, and can be used during recovery to recreate the connection. However, it is possible that some of the information is lost during the failure, if the failure occurs while it is being written. In order to recreate those connections, you must provide one implementations of the JBossJTA interface **com.arjuna.ats.jta.recovery.XAResourceRecovery** for each database that may be used by an application.

NOTE

If you are using the transactional JDBC 2.0 driver provided with JBossJTA, no additional work is necessary in order to ensure that recovery occurs.

To inform the recovery system about each of the **XAResourceRecovery** instances, specify their class names through properties. Any property found in the **properties** file, or registered at run-time, starting with the name com.arjuna.ats.jta.recovery.XAResourceRecovery is recognized as representing one of these instances. Its value is the class name, such as:
com.arjuna.ats.jta.recovery.XAResourceRecoveryOracle=com.foo.barRecovery

Additional information to be passed to the instance at creation can be specified after a semicolon:
com.arjuna.ats.jta.recovery.XAResourceRecoveryOracle=com.foo.barRecovery;myData=hello

NOTE

These properties should be in the JTA section of the **property** file.

Any errors will be reported during recovery.

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

    public boolean initialise (String p);

    public boolean hasMoreResources ();
};
```

Each method should return the following information:

initialize

After the instance is created, any additional information found after the first semicolon in the property value definition is passed to the object. The object can use this information in an implementation-specific manner.

hasMoreResources

Each **XAResourceRecovery** implementation can provide multiple **XAResource** instances. Before calling to **getXAResource**, **hasMoreResources** is called to determine whether any further connections need to be obtained. If the return value is **false**, **getXAResource** is not called called again during this recovery sweep and the instance is ignored until the next recovery scan.

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 be sufficient to create new **XAResources** instances that can be used to drive recovery.

NOTE

If you want your **XAResourceRecovery** instance to be called during each sweep of the recovery manager, 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.

4.3. ALTERNATIVE TO XARESOURCERECOVERY

The iterator-based approach that **XAResourceRecovery** uses needs to be implemented with the ability to manage states. This leads to unnecessary complexity. In JBoss Transaction Service, you can provide an implementation of the public interface, as below:

```
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 is called, and attempts recovery on each element of the array. For the majority of resource managers, you only need one **XAResource** in the array, since the **recover** method can return multiple Xids.

Unlike instances of **XAResourceRecovery** instances, which are configured via the XML properties file and instantiated by JBoss Transaction Service, instances of **XAResourceRecoveryHelper** are constructed by the application code and registered with JBoss Transaction Service by calling **XARecoveryModule.addXAResourceRecoveryHelper**.

The **initialize** method is not currently called by JBoss Transaction Service, but is provided to allow for the addition of further configuration options in later releases.

You can deregister **XAResourceRecoveryHelper** instances, after which they will no longer be called by the recovery manager. Deregistration may block for a while, if a recovery scan is in progress.

The ability to dynamically add and remove instances of **XAResourceRecoveryHelper** while the system is running is beneficial for environments where datasources may be deployed or undeployed, such as application servers. Be careful when classloading behavior in these cases.

CHAPTER 5. JDBC AND TRANSACTIONS

5.1. USING THE TRANSACTIONAL JDBC DRIVER

JBossJTA supports the construction of local and distributed transactional applications which access databases using the JDBC 2.0 APIs. JDBC 2.0 supports two-phase commit of transactions, and is similar to the XA X/Open standard. The JDBC 2.0 support is found in the com.arjuna.ats.jdbc package.

The JDBC 2.0 support has been certified with current versions of most enterprise database vendors. See http://www.jboss.com/products/platforms/application/supportedconfigurations/ for supported configurations.

5.1.1. Managing Transactions

JBossJTA needs to associate work performed on a JDBC connection with a specific transaction. Therefore, applications must use implicit transaction propagation and indirect transaction management. For each JDBC connection, JBossJTA must be able to determine the invoking thread's current transaction context.

5.1.2. Restrictions

Nested transactions are not supported by JDBC 2.0. If you try to use a JDBC connection within a subtransaction, JBossJTA throws an exception and no work is performed using that connection.

5.2. TRANSACTIONAL DRIVERS

The JBossJTA provides JDBC drivers to incorporate JDBC connections within transactions. These drivers intercept all invocations and connect them to the appropriate transactions. A given JDBC driver can only be driven by a single type of transactional driver. If the database is not transactional, ACID (atomicity, consistency, isolation, durability) properties cannot be guaranteed. Invoke the driver using the `com.arjuna.ats.jdbc.TransactionalDriver` interface, which implements the `java.sql.Driver` interface.

5.2.1. Loading drivers

You can instantiate and use the driver from within an application. For example:

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

The JDBC driver manager (`java.sql.DriverManager`) to manage driver instances by adding them to the Java system properties. The jdbc.drivers property contains a list of driver class names, separated by colons, which the JDBC driver manager loads when it is initialized.

Alternatively, you can use the `Class.forName()` method to load the driver or drivers.

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

Calling the `Class.forName()` method automatically registers the driver with the JDBC driver manager. You can also explicitly create an instance of the JDBC driver.

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

After you load the driver, it is available for making a connection with a DBMS.

5.3. CONNECTIONS

The JBossJTA provides management for transactional JDBC connections. There are some implications to using them within an application, which the developer should be aware of.

5.3.1. Making the connection

Because JBossJTA provides a new JDBC driver, application code is only lightly impacted by adding transaction support. The only thing you need to do in your code is start and end transactions.

5.3.2. JBossJTA JDBC Driver Properties

The following properties can be set and passed to the JBossJTA driver. Set them 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

> set this to `true` to cause the driver to try to create the database when it connects. This may not be supported by all JDBC 2.0 implementations.

dynamicClass

> dynamicClass: this specifies a class to instantiate to connect to the database, rather than using JNDI.

5.3.3. `XADataSources`

JDBC 2.0 connections are created from appropriate DataSources. Those connections, which participate within distributed transactions, are obtained from **XADataSources**. JBossJTA uses the appropriate DataSource when a connection to the database is made. It then obtains **XAResources** and registers them with the transaction using the JTA interfaces. The transaction service uses these **XAResources** when the transaction terminates, triggering the database to either commit or rollback the changes made via the JDBC connection.

The JBossJTA JDBC 2.0 driver can obtain **XADataSources** in one of two ways. For simplicity, it is assumed that the JDBC 2.0 driver is instantiated directly by the application.

5.3.3.1. Java Naming and Directory Interface (JNDI)

JNDI is used so that JDBC drivers can use arbitrary DataSources without knowing implementations-specific details. You can create a specific (XA)DataSource and register it with an appropriate JNDI implementation, which allows either the application or the JDBC driver to bind to and use it. Since JNDI only allows the application to see the (XA)DataSource as an instance of the interface, rather than as an instance of the implementation class, the application is not limited to only using a specific (XA)DataSource implementation.

To make the **TransactionalDriver** class use a JNDI registered **XADataSource** you need to create the **XADataSource** instance and store it in an appropriate JNDI implementation.

```
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);
```

The Context.INITIAL_CONTEXT_FACTORY property is how JNDI specifies the type of JNDI implementation to use.

The next step is for the application must pass an appropriate connection URL to the JDBC 2.0 driver.

```
Properties dbProps = new Properties();

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

TransactionalDriver arjunaJDBC2Driver = new TransactionalDriver();
Connection connection = arjunaJDBC2Driver.connect("jdbc:arjuna:jdbc/foo",
dbProps);
```

The JNDI URL must begin with **jdbc:arjuna:** in order for the **ArjunaJDBC2Driver** interface to recognize that the DataSource needs to participate within transactions and be driven accordingly.

5.3.3.2. Dynamic class instantiation

Dynamic class instantiation is not supported or recommended. Instead, use JNDI. Refer to Section 5.3.3.1, "Java Naming and Directory Interface (JNDI)" for details.

5.3.3.3. Using the connection

Once the connection has been established using the appropriate method, JBossJTA monitors all operations on the connection. If a transaction is not present when the connection is used, then operations are performed directly on the database.

You can use transaction timeouts to automatically terminate transactions if the connection has outlived its usefulness.

You can use JBossJTA connections within multiple transactions simultaneously. 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 each transaction may be using a different connection instances. With the exception of the **close** method, all operations performed on the connection at the application level use this transaction-specific connection exclusively.

JBossJTA automatically registers the JDBC driver connection with the transaction using an appropriate resource. When the transaction terminates, this resource either commits or rolls back any changes made to the underlying database using appropriate calls on the JDBC driver.

After the driver and connection are created, they 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");
```

5.3.3.4. Connection Pooling

For each user name and password, JBossJTA maintains a single instance of each connection for as long as that connection is in use. Subsequent requests for the same connection get a reference to the original connection, instead of a new instance. You can explicitly close the connection, but your request will be ignored until all users, including transactions, have either finished with the connection, or issued `close` method requests.

5.3.3.5. Reusing Connections

A very few JDBC drivers allow you to reuse a connection for multiple transactions. Most drivers require a new connection for each new transaction. By default, the JBossJTA transactional driver always obtains a new connection for each new transaction. However, if an existing connection is available and is currently unused, you can use the set the reuseconnection property to `true` on the JDBC URL.

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

5.3.3.6. Terminating the Transaction

Whenever a transaction terminates and has a JDBC connection registered with it, the JBossJTA JDBC driver instructs the database to either commit or roll back pending changes. This happens in the background, out of the purview of the application.

5.3.3.7. AutoCommit

If the AutoCommit property of the java.sql.Connection property is set to `true` for JDBC 1.0, the execution of every SQL statement is a separate top-level transaction, and it is not possible to group multiple statements to be managed within a single OTS transaction. Therefore, JBossJTA disables AutoCommit on JDBC 1.0 connections before using them. If AutoCommit is subsequently set to `true` by the application, JBossJTA raises the `java.sql.SQLException` exception.

5.3.3.8. Setting Isolation Levels

When you use the JBossJTA JDBC driver, you may need to set the underlying transaction isolation level on the XA connection. Its default value is TRANSACTION_SERIALIZABLE, but you can set this to

something more appropriate for your application by setting the com.arjuna.ats.jdbc.isolationLevel property to the appropriate isolation level in string form. Possible values include TRANSACTION_READ_COMMITTED and TRANSACTION_REPEATABLE_READ.
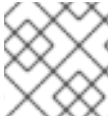
NOTE

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

CHAPTER 6. EXAMPLES

6.1. JDBC EXAMPLE

The example in Example 6.1, "JDBCTest Example" illustrates many of the points described in the JDBC chapter. Refer back to it for more information.

Example 6.1. JDBCTest Example

```
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)
        {
        }
```

```java
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(true);
            }
     catch (Exception ex)
            {
        ex.printStackTrace();
        System.exit(0);
            }
            }
     catch (Exception sysEx)
            {
        sysEx.printStackTrace();
        System.exit(0);
            }
            }
```

6.2. BASICXARECOVERY EXAMPLE FOR FAILURE RECOVERY

This class implements the **XAResourceRecovery** interface for **XAResources**. The parameter supplied in setParameters can contain arbitrary information necessary to initialize the class once created. Here, it contains the name of the property file containing database connection information, as well as the number of connections that this file knows about. Values are separated by semi-colons.

It is important to understand that this is only an example, and does not contain everything which the **XAResourceRecovery** is capable of. In real life, it is not recommended to store database connection information such as user names and passwords in a raw text file, as this example does.

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 to make it more concise.

Example 6.2. XAResourceRecovery Example

```java
/*
 * 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);
```

```java
        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)
```

```java
{
    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 = &#39;;&#39;; // delimiter for
parameters
```

The **com.arjuna.ats.internal.jdbc.recovery.JDBC2RecoveryConnection** class can create a new connection to the database using the same parameters used to create the initial connection.

CHAPTER 7. CONFIGURING JBOSSJTA

7.1. CONFIGURING OPTIONS

JTA Configuration Options and Default Values shows the configuration features with default values and relevant section numbers for more detailed information.

JTA Configuration Options and Default Values

com.arjuna.ats.jta.supportSubtransactions

Default Values: Yes/No

com.arjuna.ats.jta.jtaTMImplementation

com.arjuna.ats.jta.jtaUTImplementation

Default Values:
com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple/com.arjuna.ats.internal.jt

com.arjuna.ats.jta.xaBackoffPeriod

com.arjuna.ats.jdbc.isolationLevel

Default Values: Any supported JDBC isolation level.

com.arjuna.ats.jta.xaTransactionTimetouEnabled

Default Values: true / false

CHAPTER 8. USING JBOSSJTA WITH JBOSS ENTERPRISE APPLICATION PLATFORM

8.1. SERVICE CONFIGURATION

Most of the configuration for The JBoss Transaction Service is done using the XML files stored in the `etc` directory. Several extra attributes can be configured when it is run as a JBOSS service, however.

TransactionTimeout

> The default transaction timeout to be used for new transactions. An integer value, expressed in seconds.

StatisticsEnabled

> Determines whether or not the transaction service should gather statistical information. Viable using the `PerformanceStatistics` MBean. A Boolean value, whose default is `NO`.

PropagateFullContext

> 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 is propagated, including any parent transactions.

These attributes are specified as MBean attributes in the `jboss-service.xml` file located in the `server/all/conf` directory, See Example 8.1, "Example `jboss-services.xml`" for an example.

> **Example 8.1. Example jboss-services.xml**
>
> ```
> <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 can also be configured via the standard JBoss Transaction Service property files. These are located in the JBoss Transaction Service install location under the `etc` directory. You can edit these files manually or through JMX. Each property file is exposed using an object whose name is derived from a combination of `com.arjuna.ts.properties` and the name of the module containing the attribute to be configured An example example is `com.arjuna.ts.properties:module=arjuna`.

8.2. LOGGING

In order to make JBoss Transaction Service logging semantically consistent with JBoss Enterprise Application Platform, the `TransactionManagerService` service modifies the level of some log messages by overriding the value of the com.arjuna.common.util.logger property given in the `jbossjta-properties.xml` file. Therefore, the value of this property has no effect on the logging behavior when the transaction service is embedded in JBoss Enterprise Application Platform. By

forcing use of the `log4j_releveler` logger, the `TransactionManagerService` service causes all **INFO** level messages in the transaction code to be modified to be shown as **DEBUG** messages. As a side effect, these messages do not appear in log files if the filter level is **INFO**. All other log messages behave as normal.

8.3. THE SERVICES

The only service provided by the integration with the JBoss Enterprise Application Platform is `TransactionManagerService`. This service ensures the recovery manager is started, and binds the JBoss Transaction Service JTA transaction manager to the JNDI provider using the name `java:/TransactionManager`. 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

Uses the JBoss Transaction Service-enabled transaction manager implementation and supports distributed transactions and recovery. Configured with the `com.arjuna.ats.jbossatx.jts.TransactionManagerService` class. This is the default configuration.

Local

Uses the purely local JTA implementation. Configured using the `com.arjuna.ats.jbossatx.jta.TransactionManagerService` class.

8.4. ENSURING TRANSACTIONAL CONTEXT IS PROPAGATED TO THE SERVER

Transactions can be coordinated by a coordinator which is external to the JBoss server. To ensure that 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. Here is an example: `JRMPInvokerProxy.setTPCFactory( new com.arjuna.ats.internal.jbossatx.jts.PropagationContextManager() );`

PART II. JTS DEVELOPMENT

This section gives guidance for using the JBoss implementation of the Java Transaction Service (JTS) API to add transactional support for your enterprise applications.

CHAPTER 9. OVERVIEW

9.1. INTRODUCTION

This chapter contains a description of the use of the JBoss Transaction Service and the *Transactional Objects for Java* toolkit. The classes mentioned in this chapter are the key to writing fault-tolerant applications using transactions. After describing them, their application in a simple application is illustrated. The classes discussed in this chapter can be found in the `com.arjuna.ats.txoj` and `com.arjuna.ats.arjuna` packages.

9.2. JBOSS TRANSACTION SERVICE

In keeping with the object-oriented view, the mechanisms needed to construct reliable distributed applications are presented to programmers in an object-oriented manner. Some mechanisms, such as concurrency control and state management, need to be inherited. Others, such as object storage and transactions, are implemented as JBoss Transaction Service objects that are created and manipulated like any other object.
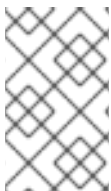
> **NOTE**
>
> When using persistence and concurrency control facilities, it is assumed that the Transactional Objects for Java (TXOJ) classes are being used. Other mechanisms are not discussed here.

JBoss Transaction Service uses object-oriented techniques to present programmers with a toolkit of Java classes which application classes can inherit to obtain desired properties, such as persistence and concurrency control. These classes form a hierarchy, part of which is shown below and which will be described later in this document.
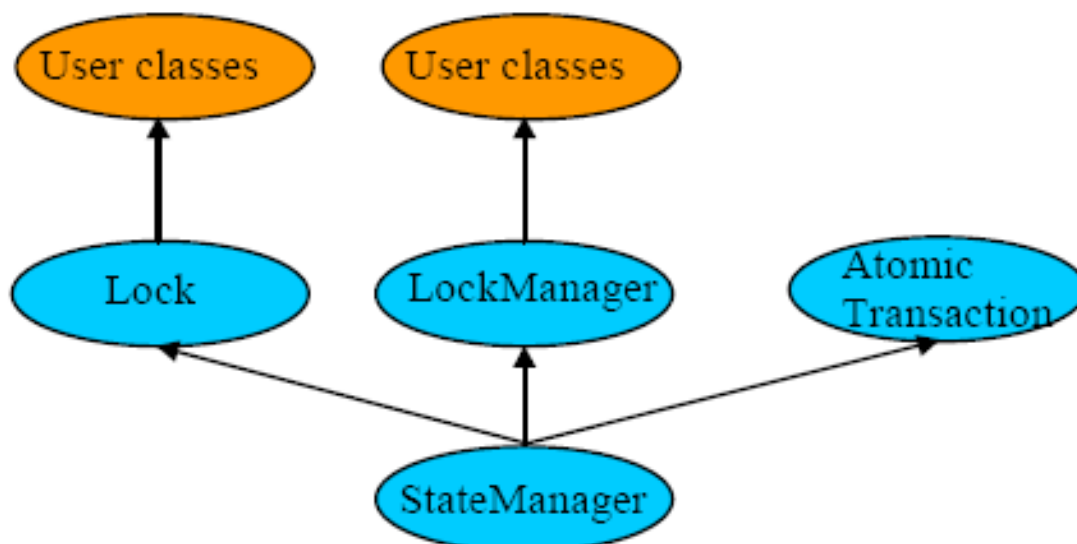
Figure 9.1. JBoss Transaction Service Class Hierarchy

The programmer is only responsible for specifying the scopes of transactions and setting appropriate locks within objects. JBoss Transaction Service and *Transactional Objects for Java*(TXOJ) ensure registration and function with the appropriate transactions, as well as crash recovery in the case of failure.

9.2.1. Saving Object States

JBoss Transaction Service remembers the state of an object. It needs this information for recovery , in which the state represents some past state of the object, and persistence, in which the state represents the final state of an object at application termination. All of these requirements are implemented using the same mechanism: the **InputObjectState** class and the **OutputObjectState** class. The classes maintain an internal array into which instances of the standard types can be contiguously packed and unpacked using **pack** and **unpack** operations. This buffer is automatically resized as required. The instances are all stored in the buffer in a standard machine-independent form called network byte order. Any other architecture-independent format, such as XDR or ASN.1), can be implemented by replacing the classes with the ones corresponding to the pack and unpack function in the required format.

9.2.2. The Object Store

The Java SecurityManager imposes some restrictions on the implementation of persistence. Therefore, the object store provided with JBoss Transaction Service uses the techniques of interface/implementation. The implementations in JBoss Transaction Service write object states to the local file system or a database, or use a client stub to implement an interface to remote services.

When persistent objects are created, they are given unique identifiers, which are actually instances of the Uid class. They can be identified within the object store by using these UIDs. States are read using the **read_committed** operation and written by the **write_committed** and **write_uncommitted** operations.

9.2.3. Recovery and persistence

The **StateManager** class is at the root of the class hierarchy, and is responsible for object activation and deactivation and object recovery. See Example 9.1, "**StateManager** Implementation".

Example 9.1. StateManager Implementation

```
public abstract class StateManager
{
public boolean activate ();
public boolean deactivate (boolean commit);

public Uid get_uid (); // object's identifier.

// methods to be provided by a derived class

public boolean restore_state (InputObjectState os);
public boolean save_state (OutputObjectState os);

protected StateManager ();
protected StateManager (Uid id);
};
```

Objects can be classified as recoverable, recoverable and persistent, or neither recoverable nor persistent.

Recoverable

StateManager attempts to generate and maintain appropriate recovery information for the object. The lifetimes of such objects do not exceed the application that created them.

Recoverable and Persistent

The lifetime of the object is greater than that of the creating or accessing application. In addition to maintaining recovery information, **StateManager** attempts to automatically **load** or **unload** any existing persistent state for the object by calling the **activate** or **deactivate** operation at the appropriate times.

Neither Recoverable Nor Persistent

No recovery information is ever kept nor is object activation or deactivation ever automatically attempted.

If an object is recoverable or recoverable and persistent, then **StateManager** invokes the **save_state** method, as part of performing the **deactivate** method, and the **restore_state**, as part of performing the **activate**, at various points during the execution of the application. The programmer must implement these methods, since **StateManager** cannot detect user-level state changes. The programmer decides which parts of an object's state should be made persistent. For example, in the case of a spreadsheet, you may not need to save all entries if some values can be recomputed instead. The Example 9.2, "**save_state** Example" example shows the **save_state** implementation for a class **Example** that has integer member variables called A, B and C.

Example 9.2. save_state Example

```java
public boolean save_state(OutputObjectState o)
{
    if (!super.save_state(o))
  return false;

    try
 {
      o.packInt(A);
      o.packInt(B);
      o.packInt(C));
}
catch (Exception e)
    {
  return false;
    }

 return true;
}
```

> **NOTE**
>
> All **save_state** and **restore_state** methods need to call **super.save_state** and **super.restore_state**, to take advantage of improvements in the crash recovery mechanisms.

9.2.4. The Life cycle of a Transactional Object for Java

A persistent object which is not in use is assumed to be in a passive state, with its state residing in an object store and activated on demand. See the Figure 9.2, "Fundamental Life cycle of a Persistent Object in TXOJ".
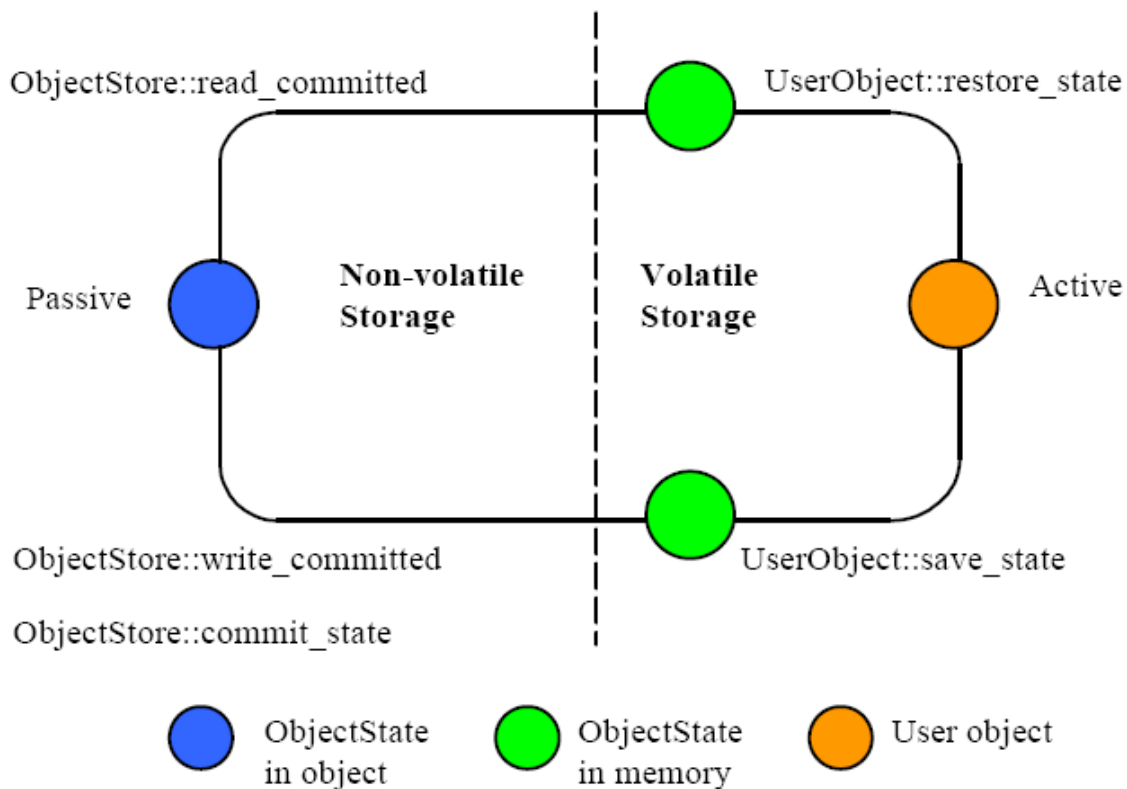
Figure 9.2. Fundamental Life cycle of a Persistent Object in TXOJ

- The object is initially passive, and is stored in the object store as an instance of the class `OutputObjectState`.

- When required by an application, the object is automatically activated by reading it from the store using a `read_committed` operation and is then converted from an `InputObjectState` instance into a fully-fledged object by the `restore_state` operation of the object.

- When the application finishes with the object, it is deactivated by converting it back into an `OutputObjectState` instance using the `save_state` operation, and is then stored back into the object store as a shadow copy using the `write_uncommitted` method. This shadow copy can be committed, overwriting the previous version, using the `commit_state` operation. The existence of shadow copies is normally hidden from the programmer by the transaction system. Object de-activation normally only occurs when the top-level transaction within which the object was activated commits.

> **NOTE**
>
> During its lifetime, a persistent object may change from passive to active and back again, many times.

9.2.5. The Concurrency Controller

The concurrency controller is implemented by the `LockManager` class, which provides sensible default behavior which the programmer can override if necessary, by the particular semantics of the class being programmed. As with the `StateManager` class and persistence, concurrency control

implementations are accessed through interfaces. The current implementations of concurrency control available to interfaces include:

- Access to remote services

- Both local disk and database implementations, where locks are written to the local file system or database to make them persistent.

- A purely local implementation, where locks are maintained within the memory of the virtual machine which created them. This implementation performs better than writing locks to the local disk, but objects cannot be shared between virtual machines. Importantly, it is a basic Java object with no requirements which can be affected by the SecurityManager

The primary API to the concurrency controller is via the **setlock** operation. By default, the runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. However, as shown in Figure 9.1, "JBoss Transaction Service Class Hierarchy", by inheriting from the **Lock** class, programmers can provide their own lock implementations with different lock conflict rules to enable *type specific concurrency control*.

Lock acquisition is, of necessity, under programmer control. Just as **StateManager** cannot determine if an operation modifies an object, **LockManager** cannot determine if an operation needs a read or write lock. Lock release, however, is under control of the system and requires no further intervention by the programmer. This ensures that the two-phase property can be correctly maintained.

```
public abstract class LockManager extends StateManager
{
    public LockResult setlock (Lock toSet, int retry, int timeout);
};
```

The **LockManager** class manages requests to set or release a lock on an object as appropriate. Since it is derived from the **StateManager** class, it can also control when some of the inherited facilities are invoked. For example, **LockManager** assumes that the setting of a write lock implies that the invoking operation must be about to modify the object, and may trigger the saving of recovery information if the object is recoverable. In a similar fashion, successful lock acquisition causes activate to be invoked.

Example 9.3. Trying to Obtain A Write Lock

```
public class Example extends LockManager
{
    public boolean foobar ()
    {
AtomicAction A = new AtomicAction;
boolean result = false;

A.begin();

if (setlock(new Lock(LockMode.WRITE), 0) == Lock.GRANTED)
    {
 /*
  * Do some work, and TXOJ will
  * guarantee ACID properties.
  */

 // automatically aborts if fails
```

```
    if (A.commit() == AtomicAction.COMMITTED)
        {
    result = true;
        }
        }
  else
      A.rollback();

  return result;
      }
}
```

9.2.6. The Transaction Protocol Engine

The transaction protocol engine is represented by the **AtomicAction** class, which uses **StateManager** to record sufficient information for crash recovery mechanisms to complete the transaction in the event of failures. It has methods for starting and terminating the transaction. For those situations where programmers need to implement their own resources, methods for registering them with the current transaction are also provided. Because JBoss Transaction Service supports subtransactions, if a transaction is begun within the scope of an already executing transaction, it is automatically nested.

NOTE

JBoss Transaction Service is multi-thread aware, allowing each thread within an application to share a transaction or execute within its own transaction. Therefore, all JBoss Transaction Service classes are also thread safe.

9.2.7. Example

The simple example below illustrates the relationships between activation, termination and commitment:

Example 9.4. Relationships Between Activation, Termination, and Commitment

```
{
      . . .
      O1 objct1 = new objct1(Name-A);/* (i) bind to "old" persistent
object A */
      O2 objct2 = new objct2();  /* create a "new" persistent object */
      OTS.current().begin();   /* (ii) start of atomic action */

      objct1.op(...);        /* (iii) object activation and invocations
*/
      objct2.op(...);
      . . .
      OTS.current().commit(true);  /* (iv) tx commits &amp; objects
deactivated */
      }      /* (v) */
```

The execution of the above code involves the following sequence of activities:

1. Creation of bindings to persistent objects. This might involve the creation of stub objects

and a call to remote objects. The above example re-binds to an existing persistent object identified by **Name-A**, and a new persistent object. A naming system for remote objects maintains the mapping between object names and locations and is described in a later chapter.

2. Start of the atomic transaction.

3. Operation invocations. As a part of a given invocation, the object implementation ensures that it is locked in read or write mode, assuming no lock conflict, and initialized, if necessary, with the latest committed state from the object store. The first time a lock is acquired on an object within a transaction, the object's state is also acquired from the object store.

4. Commit of the top-level action. This includes updating the state of any modified objects in the object store.

5. Breaking of the previously created bindings.

9.2.8. The Class Hierarchy

The principal classes which make up the class hierarchy of JBoss Transaction Service are depicted in Example 9.5, "JBoss Transaction Service Class Hierarchy".

Example 9.5. JBoss Transaction Service Class Hierarchy

```
StateManager  // Basic naming, persistence and recovery control
LockManager  // Basic two-phase locking concurrency control service
User-Defined Classes
Lock  // Standard lock type for multiple readers/single writer
User-Defined Lock Classes
AbstractRecord  // Important utility class, similar to Resource
RecoveryRecord              // handles object recovery
LockRecord  // handles object locking
RecordList  // Intentions list
other management record types
AtomicAction  // Implements transaction control abstraction
TopLevelTransaction
Input/OutputBuffer // Architecture neutral representation of an objects'
state
Input/OutputObjectState // Convenient interface to Buffer
ObjectStore  // Interface to the object storage services
```

Programmers of fault-tolerant applications need the **LockManager**, **Lock** and **AtomicAction** classes. Other classes important to a programmer are **Uid**, and **ObjectState**. Most JBoss Transaction Service classes are derived from the base class **StateManager,** which provides primitive facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and deactivation of objects, and state-based object recovery. The class **LockManager** uses the facilities of **StateManager** and **Lock** to provide the concurrency control required for implementing the serializability property of atomic actions. The implementation of atomic action facilities is supported by **AtomicAction** and **TopLevelTransaction**.

Consider a simple example. Assume that **Example** is a user-defined persistent class suitably derived

from the **LockManager**. An application containing an atomic transaction Trans accesses an object (called O) of type **Example** by invoking the operation op1 which involves state changes to O. The serialisability property requires that a write lock must be acquired on O before it is modified; thus the body of op1 should contain a call to the **setlock** operation of the concurrency controller:

```
public boolean op1 (...)
{
    if (setlock (new Lock(LockMode.WRITE) == LockResult.GRANTED)
  {
     // actual state change operations follow
     ...
  }
  }
```

The **setlock** method, provided by the **LockManager** class, performs the following functions in this case:

- Check write lock compatibility with the currently held locks, and if allowed:

- Call the **StateManager** operation **activate**, which loads, if not done already, the latest persistent state of O from the object store. Then call the **StateManager** operation **modified**, which creates an instance of either **RecoveryRecord** or **PersistenceRecord** for O, depending upon whether O is persistent or not, and inserts it into the **RecordList** of **Trans**.

- Create and insert a LockRecord instance in the RecordList of Trans.

If **Trans** is aborted some time after the lock has been acquired, the **rollback** operation of **AtomicAction** processes the **RecordList** instance associated with **Trans** by invoking an appropriate **Abort** operation on the various records. The implementation of this operation by the **LockRecord** class releases the **WRITE** lock while that of **RecoveryRecord/PersistenceRecord** restores the prior state of O.

All of the above work is automatically being performed by JBoss Transaction Service on behalf of the application programmer. The programmer only starts the transaction and sets an appropriate lock. JBoss Transaction Service and Transactional Objects for Java take care of participant registration, persistence, concurrency control and recovery.

CHAPTER 10. USING JBOSS TRANSACTION SERVICE

10.1. INTRODUCTION

This section covers JBoss Transaction Service and Transactional Objects for Java in detail, as well as how you can use it to construct transactional applications.

10.2. STATE MANAGEMENT

10.2.1. Object States

JBoss Transaction Service remembers the state of an object for the purposes of recovery and persistence. In the case of recovery, the state represents some past state of the object. When persistence is involved, the state represents the final state of an object at application termination. Since recovery and persistence include common functionality, they are both implemented using the **Input/OutputObjectState** and **Input/OutputBuffer** classes.

The **InputBuffer** class in Example 10.2, "InputBuffer" and the **OutputBuffer** classe in Example 10.1, "OutputBuffer Example" maintain an internal array into which instances of the standard Java types can be contiguously packed (unpacked) using the pack (unpack) operations. This buffer is automatically resized as required. The instances are all stored in the buffer in a standard form called *network byte order*, making them machine-independent.

Example 10.1. OutputBuffer Example

```java
public class OutputBuffer
{
    public OutputBuffer ();

    public final synchronized boolean valid ();
    public synchronized byte[] buffer();
    public synchronized int length ();

    /* pack operations for standard Java types */

    public synchronized void packByte (byte b) throws IOException;
    public synchronized void packBytes (byte[] b) throws IOException;
    public synchronized void packBoolean (boolean b) throws
IOException;
    public synchronized void packChar (char c) throws IOException;
    public synchronized void packShort (short s) throws IOException;
    public synchronized void packInt (int i) throws IOException;
    public synchronized void packLong (long l) throws IOException;
    public synchronized void packFloat (float f) throws IOException;
    public synchronized void packDouble (double d) throws IOException;
    public synchronized void packString (String s) throws IOException;
};
```

Example 10.2. InputBuffer

```java
public class InputBuffer
```

```
{
    public InputBuffer ();

    public final synchronized boolean valid ();
    public synchronized byte[] buffer();
    public synchronized int length ();

    /* unpack operations for standard Java types */

    public synchronized byte unpackByte () throws IOException;
    public synchronized byte[] unpackBytes () throws IOException;
    public synchronized boolean unpackBoolean () throws IOException;
    public synchronized char unpackChar () throws IOException;
    public synchronized short unpackShort () throws IOException;
    public synchronized int unpackInt () throws IOException;
    public synchronized long unpackLong () throws IOException;
    public synchronized float unpackFloat () throws IOException;
    public synchronized double unpackDouble () throws IOException;
    public synchronized String unpackString () throws IOException;
};
```

Example 10.3. OutputObjectState

```
class OutputObjectState extends OutputBuffer
{
    public OutputObjectState (Uid newUid, String typeName);

    public boolean notempty ();
    public int size ();
    public Uidpublic class InputBuffer
    {
 public InputBuffer ();

 public final synchronized boolean valid ();
 public synchronized byte[] buffer();
 public synchronized int length ();

 /* unpack operations for standard Java types */

 public synchronized byte unpackByte () throws IOException;
 public synchronized byte[] unpackBytes () throws IOException;
 public synchronized boolean unpackBoolean () throws IOException;
 public synchronized char unpackChar () throws IOException;
 public synchronized short unpackShort () throws IOException;
 public synchronized int unpackInt () throws IOException;
 public synchronized long unpackLong () throws IOException;
 public synchronized float unpackFloat () throws IOException;
 public synchronized double unpackDouble () throws IOException;
 public synchronized String unpackString () throws IOException;
    };
```

Example 10.4. InputObjectState

```
class InputObjectState extends InputBuffer
{
    public InputObjectState (Uid newUid, String typeName, byte[] b);

    public boolean notempty ();
    public int size ();
    public Uid stateUid ();
    public String type ();
};
```

The Example 10.4, "InputObjectState" and Example 10.3, "OutputObjectState" classes provides all the functionality of the **InputBuffer** and **OutputBuffer** classes, through inheritance. They also add two additional instance variables that signify the Uid and type of the object for which the **InputObjectState** or **OutputObjectState** instance is a compressed image. These are used when accessing the object store during storage and retrieval of the object state.

10.2.2. The Object Store

The Object Store provided with JBoss Transaction Service has a fairly restricted interface so that it can be implemented in a variety of ways. For example, object stores may reside in shared memory, in a local filesystem, or in a remote database. More complete information about the Object Stores available in JBoss Transaction Service can be found in the Appendix.

NOTE

As with all JBoss Transaction Service classes, the default Object Stores are pure Java implementations. You need to use native methods to access the shared memory and other more complex object store implementations.

All of the object stores hold and retrieve instances of the **InputObjectState** and **OutputObjectState** classes, which are named by the Uid and Type of the object that they represent. States are read using the **read_committed** method and written by the system using the **write_uncommitted** method. Normally, new object states do not overwrite old object states, but are written to the store as shadow copies. These shadows replace the original only when the **commit_state** method is invoked. All interaction with the object store is performed by JBoss Transaction Service system components as appropriate, hiding the existence of any shadow versions of objects from the programmer.

```
public class ObjectStore
{
    public static final int OS_COMMITTED;
    public static final int OS_UNCOMMITTED;
    public static final int OS_COMMITTED_HIDDEN;
    public static final int OS_UNCOMMITTED_HIDDEN;
    public static final int OS_UNKNOWN;

    /* The abstract interface */
    public abstract boolean commit_state (Uid u, String name)
  throws ObjectStoreException;
    public abstract InputObjectState read_committed (Uid u, String name)
  throws ObjectStoreException;
    public abstract boolean write_uncommitted (Uid u, String name,
```

```
        OutputObjectState os) throws ObjectStoreException;
    . . .
};
```

When a transactional object is committing, it needs to make certain state changes persistent, so that it can recover in the event of a failure and either continue to commit, or rollback. When using Transactional Objects for Java, JBoss Transaction Service manages this persistence automatically. To guarantee ACID properties, these state changes are flushed to the persistence store implementation before the transaction commits. Otherwise, the application assumes that the transaction has committed, even though the state changes may still exist within an operating system cache, vulnerable to a system failure. By default, JBoss Transaction Service flushes such state changes. As a trade-off, this behavior can impose a significant performance penalty on the application. To prevent transactional object state flushes, set the **com.arjuna.ats.arjuna.objectstore.objectStoreSync** variable to OFF.

10.2.3. StateManager

The JBoss Transaction Service class **StateManager** manages the state of an object and provides all of the basic state-management support mechanisms. **StateManager** creates and registers appropriate resources for persistence and recovery of the transactional object. If a transaction is nested, then **StateManager** propagates these resources between child transactions and their parents during the **commit** phase.

Objects in JBoss Transaction Service might be recoverable, persistent, both, or neither. If recoverable, **StateManager** tries to generate and maintain appropriate recovery information for the object, storing the information in instances of the **Input/OutputObjectState** class. The lifetimes of these objects are assumed to be shorter than the application which created them. Recoverable and persistent objects are assumed to live longer than the applications that created them, so **StateManager** loads or unloads persistent states for the object by calling the **activate** or **deactivate** method. Objects which are neither recoverable nor persistent do not have any state data stored.

```
public class ObjectStatus
{
    public static final int PASSIVE;
    public static final int PASSIVE_NEW;
    public static final int ACTIVE;
    public static final int ACTIVE_NEW;
    public static final int UNKNOWN_STATUS;
};

public class ObjectType
{
    public static final int RECOVERABLE;
    public static final int ANDPERSISTENT;
    public static final int NEITHER;
};

public abstract class StateManager
{
    public synchronized boolean activate ();
    public synchronized boolean activate (String storeRoot);
    public synchronized boolean deactivate ();
    public synchronized boolean deactivate (String storeRoot, boolean
commit);
```

```
    public synchronized void destroy ();

    public final Uid get_uid ();

    public boolean restore_state (InputObjectState, int ObjectType);
    public boolean save_state (OutputObjectState, int ObjectType);
    public String type ();
    . . .

  protected StateManager ();
    protected StateManager (int ObjectType, ObjectName attr);
    protected StateManager (Uid uid);
    protected StateManager (Uid uid, ObjectName attr);
    . . .

  protected final void modified ();
    . . .
};

public class ObjectModel
{
    public static final int SINGLE;
    public static final int MULTIPLE;
};
```

If an object is recoverable or persistent, **StateManager** invokes the **save_state** method during the **deactivation** method. The **restore_state** is called during the **activate**. The **type** is called at various points during the execution of the application. The programmer must implement these methods, since **StateManager** does not have access to a runtime description of the layout of an arbitrary Java object in memory. However, the capabilities provided by **InputObjectState** and **OutputObjectState** classes simplify the writing of these routines. For example, the **save_state** implementation for a class **Example** that had member variables called A, B and C might adhere to Example 10.5, "**save_state** Example"

Example 10.5. save_state Example

```
public boolean save_state ( OutputObjectState os, int ObjectType )
{
    if (!super.save_state(os, ObjectType))
  return false;

    try
  {
    os.packInt(A);
    os.packString(B);
    os.packFloat(C);

    return true;
  }
    catch (IOException e)
  {
    return false;
  }
  }
```

To support crash recovery for persistent objects, all `save_state` and `restore_state` methods of user objects must call `super.save_state` and `super.restore_state`.

> **NOTE**
>
> The `type` method determines the location in the object store where the state of instances of that class will be saved and ultimately restored. This can be any valid string. However, avoid using the hash character #, which is reserved for special directories required by JBoss Transaction Service.

The `get_uid` method of `StateManager` provides read-only access to an object's internal system name. The value of the internal system name can only be set when an object is created,by providing it as an explicit parameter or by generating a new identifier when the object is created.

The `destroy` method removes the object's state from the object store. This is an atomic operation, which only removes the state if its invoking transaction commits. The programmer must guarante eexclusive access to the object before invoking this operation.

Since object recovery and persistence have complimentary requirements, the `StateManager` class combines the management of both into a single mechanism. That is, it uses instances of the class Input/OutputObjectState both for recovery and persistence purposes. An additional argument passed to the `save_state` and `restore_state` operations allows the programmer to determine the purpose for which any given invocation is being made thus allowing different information to be saved for recovery and persistence purposes.

10.2.4. Object Models

JBoss Transaction Service supports two models for objects. Implementation of the state and concurrency controls depend on which model is used.

Single

The application only contains a single copy of the object The object resides within a single JVM, and all clients must address their invocations to this server. The single model provides better performance, but creates a single point of failure. In a multi-threaded environment, the object may not be protected from corruption if a single thread fails.

Single Object Model
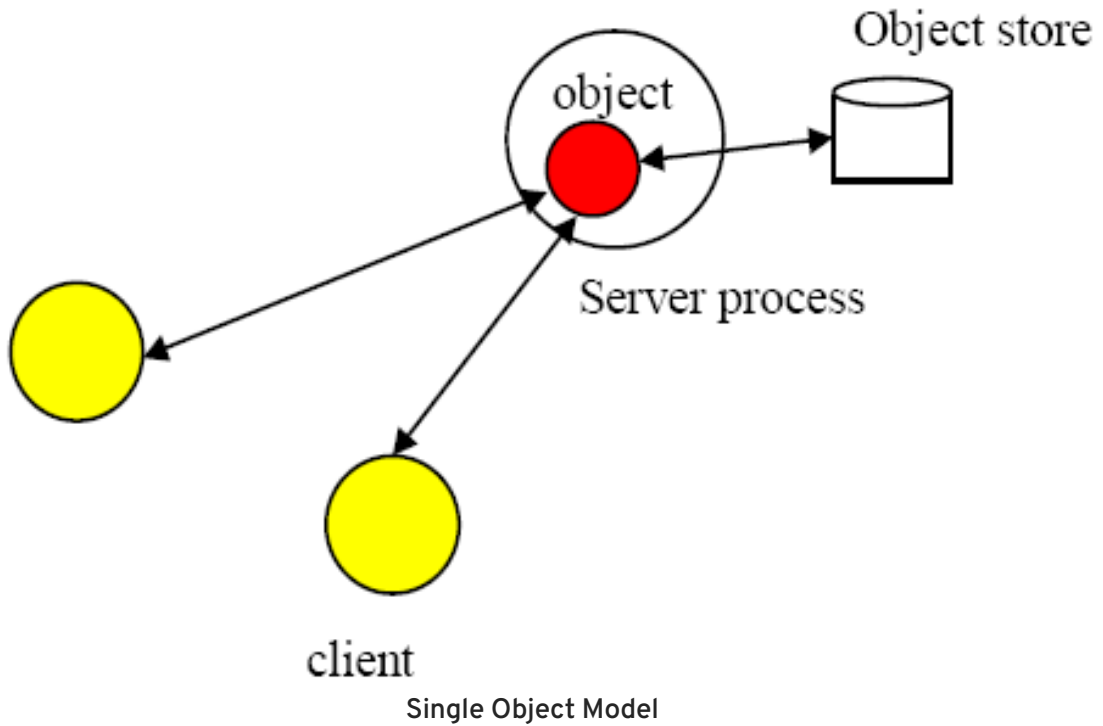
Multiple

Logically, a single instance of the object exists. Copies of the object are distributed across multiple JVMs. Performance suffers compared to the single model, better failure isolation is achieved.

Multiple Object Model
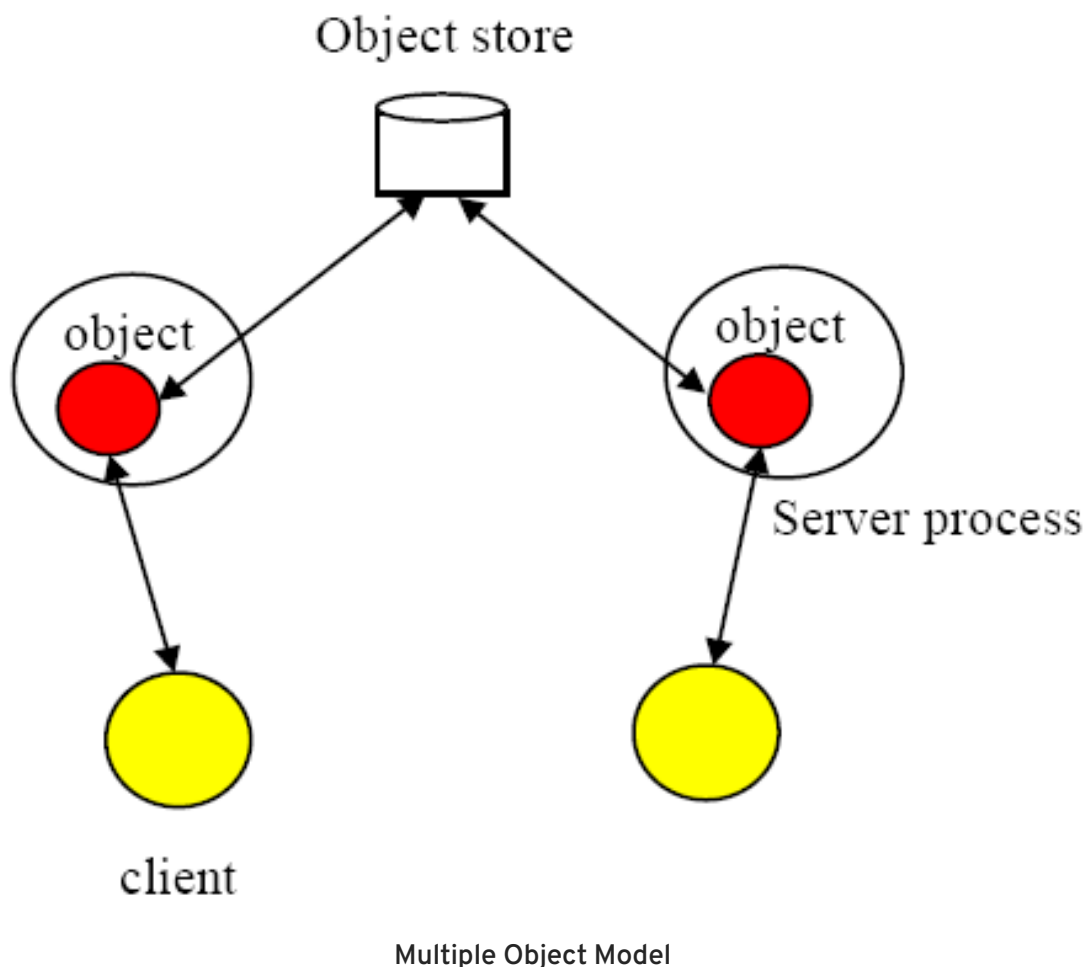
The *single* model is the default. You can override this on a per-object basis by providing an appropriate instance of the **com.arjuna.ats.arjuna.gandiva.ObjectName** class when you create your object.

> **NOTE**
>
> You can change the model before any instantiation of the object There is no need for it to remain the same during the object's lifetime.

Use the following method to provide a suitable **ObjectName** class. Refer to Example 10.6, "Object Models" for an example.

1. Create a new instance of **ObjectName**.

2. Set the object model attribute using the **com.arjuna.ats.arjuna.ArjunaNames.StateManager_objectModel()** name.

Example 10.6. Object Models

```
{
    ObjectName attr = new ObjectName("SNS:myObjectName");

    attr.setLongAttribute(ArjunaNames.StateManager_objectModel(),
     ObjectModel.SINGLE);

    AtomicObject obj = new AtomicObject(ObjectType.ANDPERSISTENT, attr);
}
```

10.2.5. JBoss Transaction Service Method Reference

The JBoss Transaction Service class **StateManager** manages the state of objects and provides all of the basic support mechanisms required for recovery, persistence, or both. Some operations must be defined by you. These operations are: **save_state**, **restore_state**, and **type**.

boolean save_state(OutputObjectState *state*, int *ObjectType*)

Invoked to save the state of an object for future use, for recovery or persistence. The **ObjectType** parameter indicates the reason for invocation. This allows you to save different pieces of information into the **OutputObjectState** supplied as the first parameter, depending on whether recovery or persistence is desired. For example, pointers to other JBoss Transaction Service objects may be saved as pointers for recovery, but as UIDs for persistence. The **OutputObjectState** class provides convenient operations, so that you can save instances of all of the basic types in Java. To support crash recovery for persistent objects all **save_state** methods need to call **super.save_state**.

> **NOTE**
>
> The **save_state** method assumes that an object is internally consistent and that all variables saved have valid values. Write and test your code to be sure this is true.

boolean restore_state(InputObjectState *state*, int *ObjectType*)

Restores an object to the specified state. The second parameter allows different interpretations of the supplied state. To support crash recovery for persistent objects all **restore_state** methods need to call **super.restore_state**.

String type ()

The JBoss Transaction Service persistence mechanism needs a way to determine the type of an object as a string, so that it can save and restore the state of the object. By convention, the position of the class in the hierarchy is used. For example, **StateManager/LockManager/Object**.
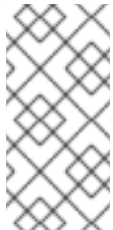
> **NOTE**
>
> The **type** method determines the location of the state of instanes of a specified class are saved into the object store. This can actually be any valid string. However, avoid using the hash character #, which is reserved for special directories required by JBoss Transaction Service.

10.2.6. Example

Example 10.7, "Saving and Restoring an Object's State" shows a basic **Array** class derived from the **StateManager** class. To illustrate saving and restoring of an object's state, the **highestIndex** variable keeps track of the highest element of the array that has a non-zero value.

Example 10.7. Saving and Restoring an Object's State

```
public class Array extends StateManager
{
    public Array ();
    public Array (Uid objUid);
    public void finalize ( super.terminate(); };

    /* Class specific operations. */

    public boolean set (int index, int value);
    public int get (int index);

    /* State management specific operations. */

    public boolean save_state (OutputObjectState os, int ObjectType);
    public boolean restore_state (InputObjectState os, int ObjectType);
    public String type ();

    public static final int ARRAY_SIZE = 10;

    private int[] elements = new int[ARRAY_SIZE];
    private int highestIndex;
};
```

The **save_state**, **restore_state** and **type** operations can be defined as follows:

```
/* Ignore ObjectType parameter for simplicity */

public boolean save_state (OutputObjectState os, int ObjectType)
```

```
{
    if (!super.save_state(os, ObjectType))
 return false;

    try
 {
     packInt(highestIndex);

     /*
      * Traverse array state that we wish to save. Only save active
elements
      */

     for (int i = 0; i <= highestIndex; i++)
  os.packInt(elements[i]);

     return true;
 }
    catch (IOException e)
 {
     return false;
 }
}

public boolean restore_state (InputObjectState os, int ObjectType)
{
    if (!super.restore_state(os, ObjectType))
 return false;

    try
 {
     int i = 0;

     highestIndex = os.unpackInt();

     while (i < ARRAY_SIZE)
  {
      if (i <= highestIndex)
   elements[i] =  os.unpackInt();
      else
   elements[i] = 0;
      i++;
  }

     return true;
 }
    catch (IOException e)
 {
     return false;
 }
}

public String type ()
{
    return '/StateManager/Array';
}
```

10.3. LOCK MANAGEMENT AND CONCURRENCY CONTROL

Concurrency control information within JBoss Transaction Service is maintained by locks. Some of these locks need to be used by multiple objects in different processes. They can be held in a lock store, similar to the object store used for state information. The lock store used with JBoss Transaction Service has a restricted interface which allows flexibility with regard to implementation. Lock stores can be implemented in shared memory, on the Unix file system in several different formats, or as a remotely accessible store.

> **NOTE**
>
> As with all JBoss Transaction Service classes, the default lock stores are pure Java implementations. If you want to use more complex lock implementations, you must use native methods.

Example 10.8. Example LockStore Class

```
public class LockStore
{
    public abstract InputObjectState read_state (Uid u, String tName)
 throws LockStoreException;

    public abstract boolean remove_state (Uid u, String tname);
    public abstract boolean write_committed (Uid u, String tName,
        OutputObjectState state);
};
```

10.3.1. Selecting a Lock Store Implementation

JBoss Transaction Service supports several different object store implementations. If the object model being used is *Single*, no lock store is required for maintaining locks, because the information about the object is not exported from it. However, if you use the *Multiple* model, different run-time environments may need to share concurrency control information. You can specify the implementation type of the lock store to use for all objects within a given execution environment using the `com.arjuna.ats.txoj.lockstore.lockStoreType` property. This variable can be either:

BasicLockStore

This is an in-memory implementation which does not include support for sharing of stored information between execution environments. You can extend it to include this functionality, if needed.

BasicPersistentLockStore

This is the default implementation. It stores locking information within the local file system. Execution environments that share the same file store can share concurrency control information. The root of the file system into which locking information is written is the `LockStore/` directory within the JBoss Transaction Service installation directory. To override this location, set the `com.arjuna.ats.txoj.lockstore.lockStoreDir` property accordingly, or include the location in the `CLASSPATH`:

How to Override the lockStoreDir Property

- **java -D
  com.arjuna.ats.txoj.lockstore.lockStoreDir=/var/tmp/LockStore
  myprogram**

- **java –classpath $CLASSPATH;/var/tmp/LockStore myprogram**

10.3.2. LockManager

The concurrency controller is implemented by the class **LockManager**, which provides sensible default behavior that you can override if necessary. The **setlock** method is the primary interface to the concurrency controller. By default, the JBoss Transaction Service runtime system enforces strict two-phase locking, following a *multiple reader, single writer* policy on a per-object basis. You, as the programmer, control lock acquisition, since the **LockManager** class cannot predict whether an operation needs a read or write lock. Lock release, however, is normally under control of the system, requiring no action by the programmer.

The **LockManager** class manages requests to set a lock on an object or to release a lock. However, since it is derived from **StateManager**, it can also control invocation of some of the inherited facilities. For example, if a request to set a write lock is granted, then **LockManager** invokes the **modified** method directly, since setting a write lock implies that the invoking method is about to modify the object. This may cause recovery information to be saved, if the object is recoverable. Successful lock acquisition also triggers invocation of the **activate**.

Therefore, **LockManager** activates and deactivates persistent objects, and also registers **Resources** used for managing concurrency control. By driving the **StateManager** class, it also registers **Resources** for persistent and recoverable state manipulation and object recovery. You only set the appropriate locks, start and end transactions, and extend the **save_state** and **restore_state** methods of the **StateManager** class.

Example 10.9. LockResult Example

```
public class LockResult
{
    public static final int GRANTED;
    public static final int REFUSED;
    public static final int RELEASED;
};

public class ConflictType
{
    public static final int CONFLICT;
    public static final int COMPATIBLE;
    public static final int PRESENT;
};

public abstract class LockManager extends StateManager
{
    public static final int defaultTimeout;
    public static final int defaultRetry;
    public static final int waitTotalTimeout;

    public synchronized int setlock (Lock l);
```

```
    public synchronized int setlock (Lock l, int retry);
    public synchronized int setlock (Lock l, int retry, int sleepTime);
    public synchronized boolean releaselock (Uid uid);

    /* abstract methods inherited from StateManager */

    public boolean restore_state (InputObjectState os, int ObjectType);
    public boolean save_state (OutputObjectState os, int ObjectType);
    public String type ();

    protected LockManager ();
    protected LockManager (int ObjectType, ObjectName attr);
    protected LockManager (Uid storeUid);
    protected LockManager (Uid storeUid, int ObjectType, ObjectName
attr);
    . . .
};
```

You need to pass the type of lock required and the number of retries to acquire the lock,as parameters to the **setlock** method. The type is either **READ** or **WRITE**. If a lock conflict occurs, one of the following scenarios will take place:

- If the retry value is equal to **LockManager.waitTotalTimeout**, the thread which called the **setlock** method is blocked until the lock is released, or the total timeout specified has elapsed. In the case of a time-out, a value of **REFUSED** is returned.

- If the lock cannot be obtained initially,**LockManager** retries the specified number of times, waiting for the specified timeout value between each failed attempt. The default is 100 attempts, each attempt being separated by a 0.25 seconds delay.

If a lock conflict occurs, the lock request is timed out, to prevent deadlocks. A full deadlock detection scheme is not provided. If the requested lock is obtained, the **setlock** method returns a value of **GRANTED**. Otherwise, a value of **REFUSED** is returned. You need to ensure that the remainder of the code for an operation is only executed if a lock request is granted. Refer to Example 10.10, "**setlock** Example" for a working example.

Example 10.10. setlock Example

```
res = setlock(new Lock(WRITE), 10);
// Attempts to set a write
// lock 11 times (10 retries)
// before giving up.

res = setlock(new Lock(READ), 0);
// Attempts to set a read lock
// 1 time (no retries) before
// giving up.

res = setlock(new Lock(WRITE);
// Attempts to set a write lock
// 101 times (default of 100
// retries) before giving up.
```

The concurrency control mechanism is integrated into the atomic action mechanism to ensure that as locks are granted on an object, appropriate information is registered with the currently running atomic action. This guarantees that the locks are released at the correct time, and removes the need to explicitly free locks which were acquired within atomic actions. However, if locks are acquired on an object outside of the scope of an atomic action, you must use the **releaselock** method to release the locks.

10.3.3. Locking policy

Locks in JBoss Transaction Service are not special system types. They are, instead, instances of other JBoss Transaction Service objects. The **Lock** class is derived from **StateManager** so that locks can be made persistent and can be named in a simple way. Furthermore, the **LockManager** class does not know about the semantics of the actual policy for granting lock requests. Instances of the **Lock** class maintain this information, and provide the **conflictsWith** method, which **LockManager** uses to determine whether two locks conflict. This separation allows you to derive new lock types from the basic **Lock** class and provides appropriate definitions of the conflict operations, allowing enhanced levels of concurrency.

```
public class LockMode
{
    public static final int READ;
    public static final int WRITE;
};

public class LockStatus
{
    public static final int LOCKFREE;
    public static final int LOCKHELD;
    public static final int LOCKRETAINED;
};

public class Lock extends StateManager
{
    public Lock (int lockMode);

    public boolean conflictsWith  (Lock otherLock);
    public boolean modifiesObject ();

    public boolean restore_state (InputObjectState os, int ObjectType);
    public boolean save_state (OutputObjectState os, int ObjectType);
    public String type ();
    . . .
};
```

The **Lock** class provides a **modifiesObject** method, which **LockManager** uses to determine a call or method is needed to grant a locking request. This allows locking modes other than simple read and write to be supported. The supplied **Lock** class supports the traditional multiple reader/single writer policy.

10.3.4. Object construction and destruction

JBoss Transaction Service objects can be either recoverable, persistent, both, or neither. Also, each object has a unique internal name. These attributes can only be set when that object is constructed. Therefore, **LockManager** provides two protected constructors for use by derived classes, each of

which fulfils a distinct purpose:

LockManager ()

> Allows the creation of new objects, which have no prior state.

LockManager(int *ObjectType*, ObjectName *attr*)

> Allows the creation of new objects, which have no prior state. The *ObjectType* parameter denotes whether an object is **recoverable**, recoverable and persistent (indicated by **ANDPERSISTENT**) or neither (**NEITHER**). If an object is marked as being persistent, its state is stored in one of the object stores. The *shared* parameter only has meaning if the object is **RECOVERABLE.**; If **attr** is not null and the object model is **SINGLE** (the default behavior), then the recoverable state of the object is maintained within the object itself. Otherwise, the state of the object is stored in an in-memory object store between atomic actions.

> Constructors for new persistent objects should make use of atomic actions within themselves. This will ensure that the state of the object is automatically written to the object store either when the action in the constructor commits, or, if an enclosing action exists, when the appropriate top-level action commits.

LockManager(Uid *objUid*)

> Allows access to the existing persistent object named in the *objUid* parameter. The object's prior state, which is identified by the value of the *objUid* parameter), is loaded from an object store automatically.

LockManager(Uid *objUid*, ObjectName *attr*)

> Allows access to the existing persistent object named in the *objUid* parameter. The object's prior state, which is identified by the value of the *objUid*, is loaded from an object store automatically. If the *attr* parameter is not null, and the object model is **SINGLE** (the default behaviour), then the object is not reactivated at the start of each top-level transaction.

The destructor of a programmer-defined class needs to invoke the inherited operation `terminate`, to inform the state management mechanism that the object is about to be destroyed. Otherwise, unpredictable results may occur.

Because the **LockManager** class inherits from **StateManager**, it passes any supplied **ObjectName** instances to the **StateManager** class. As such, you can set the **StateManager** object model as described earlier.

CHAPTER 11. GENERAL TRANSACTION ISSUES

11.1. ADVANCED TRANSACTION ISSUES WITH JBOSS TRANSACTION SERVICE

Transactions are used by both application programmers and class developers. You can make entire operations, or parts of operations, transactional. This chapter covers some of the more subtle issues involved with using transactions in general and, some particulars about JBoss Transaction Service.

11.1.1. Checking Transactions

In a multi-threaded application, multiple threads may be associated with a transaction during its lifetime, sharing the same context. Also, if one thread terminates a transaction, other threads may still be active within it. In a distributed environment, it is difficult to guarantee that no threads need a transaction when it is terminated. By default, JBoss Transaction Service issues a warning if a thread terminates a transaction when other threads are using it. However, it still allows the transaction to be terminated. Another possible behavior is to block terminating thread until all other threads have disassociated themselves from the transaction context. Therefore, JBoss Transaction Service provides the **com.arjuna.ats.arjuna.coordinator.CheckedAction** class, which allows you to override the thread/transaction termination policy. Each transaction has an instance of this class associated with it, and you can provide your own implementations on a per-transaction basis.

```
public class CheckedAction
{
    public CheckedAction ();

    public synchronized void check (boolean isCommit, Uid actUid,
        BasicList list);
};
```

When a thread tries to terminate a transaction and there are active threads within it, the system invokes the **check** method on the transaction's **CheckedAction** object. The parameters to the check method are:

isCommit

   Indicates whether the transaction is in the process of committing or rolling back.

actUid

   The transaction identifier.

list

   A list of all of the threads currently active within this transaction.

When the **check** method returns, the transaction is terminated. The state of the transaction may have changed from when the **check** method was called.

11.1.2. Gathering Statistics

By default, the JBoss Transaction Service does not maintain any historical information about transactions. You can turn on history tracking by setting the **com.arjuna.ats.arjuna.coordinator.enableStatistics** property variable to **YES**. This

information includes the number of transactions created, and the outcomes of the transactions. You can request this information during the execution of a transactional application via the `com.arjuna.TxCore.Atomic.TxStats` class, as in Example 11.1, "Transaction Statistics".

Example 11.1. Transaction Statistics

```java
public class TxStats
{

    // Returns the number of transactions (top-level and nested)
    // created so far.

    public static int numberOfTransactions ();

    // Returns the number of nested (sub) transactions created so far.

    public static int numberOfNestedTransactions ();

    // Returns the number of transactions which have terminated with
    // heuristic outcomes.

    public static int numberOfHeuristics ();

    // Returns the number of committed transactions.

    public static int numberOfCommittedTransactions ();

    // Returns the number of transactions which have rolled back.

    public static int numberOfAbortedTransactions ();

}
```

11.1.3. Last resource commit optimization

In some cases, you may need enlist participants that are not *two-phase commit aware* into a two-phase commit transaction. If there is only a single resource, you do not need two-phase commit. If the transaction contains multiple transactions, the *Last Resource Commit Optimization* (LRCO) becomes relevant. A single resource that is one-phase aware can only commit or roll back, with no prepare phase. Such a resource may be enlisted in a transaction with two-phase commit aware resources. The coordinator treats the one-phase aware resource slightly differently, by executing the prepare phase on all other resource first. Then, the one-phase aware transaction needs to commit the transaction, the coordinator passes control to it. If it commits, the coordinator logs the decision to commit and attempts to commit the other resources as well.

To use the LRCO, your participant must implement the `com.arjuna.ats.arjuna.coordinator.OnePhase` interface and be registered with the transaction through the `BasicAction.add` method. Since this operation expects instances of `AbstractRecord`, you need to create an instance of the `com.arjuna.ats.arjuna.LastResourceRecord` class and pass your participant as the constructor parameter, as shown in the Example 11.2, "BasicAction.add Example".

Example 11.2. BasicAction.add Example

```
try
    {
boolean success = false;
AtomicAction A = new AtomicAction();
OnePhase opRes = new OnePhase();  // used OnePhase interface

System.err.println("Starting top-level action.");

A.begin();
A.add(new LastResourceRecord(opRes));
A.add(new ShutdownRecord(ShutdownRecord.FAIL_IN_PREPARE));

A.commit();
    }
```

11.1.4. Nested Transactions

You do not need to do anything special to nest transactions. If an action is begun while another action is running, it is automatically nested. This provides a modular structure to applications,meaning that the programmer of the object does not need to be concerned about whether the applications to use the object are also transactional.

If a nested action is aborted then all of its work is rolled back. However, strict two-phase locking means that any locks the terminating transaction holds are retained until the top-level action commits or aborts. If a nested action commits, the work it has performed is only committed by the system if the top-level action commits. If the top-level action aborts, all the work is rolled back.

Committing or aborting a nested action does not automatically affect the outcome of its parent action. You can choose to implement this behavior programmatically, controlling the way faults are contained or work is undone, for instance.

11.1.5. Asynchronously Committing a Transaction

By default, JBoss Transaction Service executes the `commit` protocol of a top-level transaction in a synchronous, single-threaded manner. All registered resources are directed to prepare in order by a single thread, and then to commit or rollback. This has several possible disadvantages.

- Often, the `prepare` operating is logically be invoked in parallel on each resource. The disadvantage is that if an early resource in the list of registered resource forces a rollback during `prepare`, many unnecessary `prepare` operations may have been performed.

- If your application does not need heuristic reporting, the second phase of the `commit` protocol can be called asynchronously, since its success or failure is not important.

Therefore, JBoss Transaction Service provides runtime options to enable two threading optimizations, by setting specific variables.

com.arjuna.ats.arjuna.coordinator.asyncPrepare

If set to **YES**, a separate thread is created during the **prepare** phase, for each registered participant within the transaction.

com.arjuna.ats.arjuna.coordinator.asyncCommit

If set to **YES**, a separate thread is created to complete the second phase of the transaction if you do not need information about heuristics outcomes.

11.1.6. Independent Top-Level Transactions

In addition to normal top-level and nested transactions, JBoss Transaction Service also supports independent top-level actions, which you can use to relax strict serializability in a controlled way. You can execute an independent top-level action from anywhere within another transaction, and it behaves exactly like a normal top-level action. Its results are made permanent when it commits and are not undone if any of its parent actions abort.
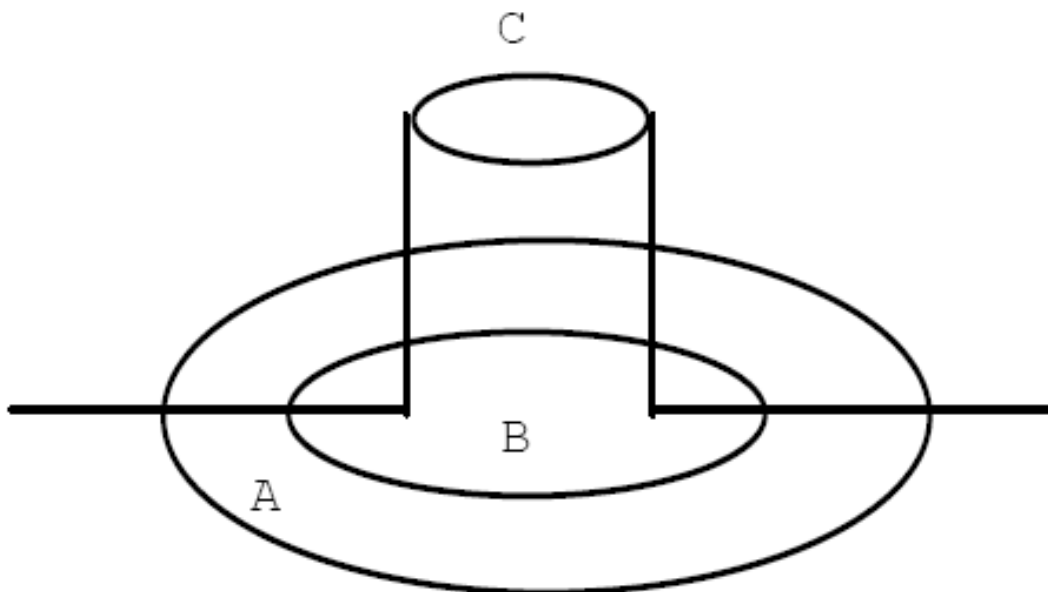
Figure 11.1. Independent Top-Level Actions

Figure 11.1, "Independent Top-Level Actions" shows a typical nesting of transactions, where action B is nested within action A. Transaction C is logically nested within action B, because its `Begin` operation is invoked while B is active. However, because it is an independent top-level action, it `commit`s or `abort`s independently of the other actions within the structure. Because of the nature of independent top-level actions, use them with caution and only after careful testing and observation.

You can use top-level actions within an application by declaring and using instances of the `TopLevelTransaction` class, and using them in exactly the same way as other transactions.

11.1.7. Transactions Within the `save_state` and `restore_state` Methods

Exercise caution when you write the **save_state** and **restore_state** methods. Make sure not to start any atomic actions, either explicitly in the method, or implicitly through use of some other operation. The reason for this caution is that JBoss Transaction Service may invoke the **restore_state** method when it commits, resulting in an attempt to execute a transaction during the **commit** or **abort** phase of another action. This might violate the atomicity properties of the action being committed or aborted, so it is strongly discouraged.

11.1.8. Example

The code in Example 11.3, "Nested Transaction Example" is based on the Example 10.7, "Saving and Restoring an Object's State" example presented earlier, and implements the `set` and `get` methods. The code is simplified, and ignores error conditions and exceptions.

Example 11.3. Nested Transaction Example

```java
public boolean set (int index, int value)
{
    boolean result = false;
    AtomicAction A = new AtomicAction();

    A.begin();

    // We need to set a WRITE lock as we want to modify the state.

    if (setlock(new Lock(LockMode.WRITE), 0) == LockResult.GRANTED)
 {
      elements[index] = value;
      if ((value > 0) && (index > highestIndex))
  highestIndex = index;
      A.commit(true);
      result = true;
 }
    else
 A.rollback();

    return result;
}

public int get (int index)  // assume -1 means error
{
    AtomicAction A = new AtomicAction();

    A.begin();

    // We only need a READ lock as the state is unchanged.

    if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
 {
       A.commit(true);

       return elements[index];
 }
    else
 A.rollback();

    return -1;
}
```

11.1.9. Garbage Collecting Objects

Java objects are deleted by the garbage collector when they are no longer needed. Caution must be used when deleting an object that is currently under the control of a transaction. If the object is being manipulated within a transaction, the transaction controls what happens to it. Therefore, regardless of

the references to a transactional object maintained by an application, JBoss Transaction Service always retains its own references to make sure that the object is not deleted by the garbage collector until all involved transactions have terminated.

11.1.10. Transaction Timeouts

By default, transactions live until the application which created them deletes them, or a failure occurs. However, you can to set a timeout on a per-transaction basis, causing transactions which do not terminate before the timeout expires to be rolled back. The timeout value is expressed in seconds.

In JBoss Transaction Service, the timeout value is provided as a parameter to the `AtomicAction` constructor. If the default value of `AtomicAction.NO_TIMEOUT` is provided, the transaction is never automatically timed out. Any other positive value represents the number of seconds to wait for the transaction to terminate. A value of `0` is interpreted as a global default timeout, which you can provide using the property `com.arjuna.ats.arjuna.coordinator.defaultTimeout`. This property's default value is `60`, or one minute.

When a top-level transaction is created with a non-zero timeout, it will be rolled back if it times out. **JBossTS** uses a separate reaper thread to monitor all locally created transactions, forcing them to roll back if their timeouts elapse. To prevent the reaper thread from consuming application time, it only runs periodically. The default checking period is 120000 milliseconds, but you can override it by setting the com.arjuna.ats.arjuna.coordinator.txReaperTimeout property to another value, in microseconds. Alternatively, if the com.arjuna.ats.arjuna.coordinator.txReaperMode property is set to **DYNAMIC**, the transaction reaper wakes up whenever a transaction times out. This has the advantage of terminating transactions early, but may continually reschedule the reaper thread.

If the timeout value is `0` for a top-level transaction, or no timeout is specified, JBoss Transaction Service does not impose any timeout on the transaction, and it is allowed to run indefinitely. You can override this default timeout by setting the com.arjuna.ats.arjuna.coordinator.defaultTimeout property.

CHAPTER 12. HINTS AND TIPS

12.1. GENERAL TIPS

12.1.1. Using Transactions in Constructors

Examples throughout this manual use transactions in the implementation of constructors for new persistent objects. This it guarantees correct propagation of the state of the object to the object store. The state of a modified persistent object is only written to the object store when the top-level transaction commits. Thus, if the constructor transaction is top-level and it commits, the newly-created object is written to the store, and becomes available immediately. However, if the constructor transaction commits but is nested because some another transaction, which was started prior to object creation, is running, then the state is only written if all of the parent transactions commit.

On the other hand, if the constructor does not use transactions, inconsistencies may arise in the system. For example, if no transaction is active when the object is created, its state is not saved to the store until the next time the object is modified under the control of a transaction.

Example 12.1. Transactions causing System Inconsistencies

```
AtomicAction A = new AtomicAction();
Object obj1;
Object obj2;

obj1 = new Object();      // create new object
obj2 = new Object("old"); // existing object

A.begin(0);
obj2.remember(obj1.get_uid()); // obj2 now contains reference to obj1
A.commit(true);      // obj2 saved but obj1 is not
```

Here, the two objects are created outside of the control of the top-level action A. **obj1** is a new object. **obj2** is an old existing object. When the **remember** method of **obj2** is invoked, the object is activated and the Uid of **obj1** is known. Since this action commits, the persistent state of **obj2** may now contain the Uid of **obj1**. However, the state of **obj1** itself has not been saved, since it has not been manipulated under the control of any action. In fact, unless it is modified under the control of some action later in the application, it will never be saved. However, if the constructor would have used an atomic action, the state of **obj1** would have automatically been saved at the time it was constructed, preventing this inconsistency.

12.1.2. More on the `save_state` and `restore_state` Methods

JBoss Transaction Service may invoke the user-defined **save_state** method of an object at any time during the lifetime of an object, including during the execution of the body of the object's constructor. This is especially true if it uses atomic actions. All of the variables saved by **save_state** are correctly initialized.

Use caution when writing the **save_state** and **restore_state** methods, to ensure that no transactions are explicitly or implicitly started. The reason is that JBoss Transaction Service may invoke the **restore_state** method as part of its **commit** processing, causing the execution of an atomic transaction during the **commit** or **abort** phase of another transaction. This may violate the atomicity properties of the transaction being committed or aborted, so it is discouraged.

To support crash recovery for persistent objects, all `save_state` and `restore_state` methods of user objects need to call the `super.save_state` and `super.restore_state`.

12.1.3. Packing Objects

All of the basic types of Java, such as int and long, can be saved and restored from an **InputObjectState** or **OutputObjectState** instances by using the **pack** and **unpack** methods provided by **InputObjectState** and **OutputObjectState**. However, you should handle packing and unpacking objects differently, because packing objects brings in the additional problems of aliasing. Alias means that two different object references may actually refer to the same item. See the Example 12.2, "Aliasing Problem with Packing Objects".

Example 12.2. Aliasing Problem with Packing Objects

```
public class Test
{
    public Test (String s);
    ...
 private String s1;
    private String s2;
};

public Test (String s)
{
    s1 = s;
    s2 = s;
}
```

Here, both s1 and s2 point to the same string, and a naive implementation of the `save_state` method has the potential to copy the string twice. From the perspective of the `save_state` method, this is merely inefficient. However, it would cause the `restore_state` method to unpack the two strings into different areas of memory, destroying the original aliasing information. JBoss Transaction Service packs and unpacks separate object references.

12.2. DIRECT USE OF THE STATEMANAGER CLASS

The examples throughout this manual derive user classes from the **LockManager** class. There are two reasons for this.

- Most importantly, the serializability constraints of atomic actions require it.

- It reduces the need for programmer intervention.

However, if you only require access to the persistence and recovery mechanisms of JBoss Transaction Service, you can directly derive a user class from **StateManager**.

Classes derived directly from **StateManager** must make use of its state management mechanisms explicitly, rather than relying on **LockManager**. You need to make appropriate use of the `activate`, `deactivate`, and `modified` methods, since the constructors of **StateManager** are effectively identical to those of **LockManager**.

12.2.1. The `activate` Method

```
boolean activate ()
boolean activate (String storeRoot)
```

The `activate` method loads an object from the object store. The object's UID needs to already be set via the constructor, and the object must exist in the store. If the object is successfully read, the `restore_state` method is called to build the object in memory. The `activate` method operates such that once an object has been activated, further calls are ignored. The parameter represents the root name of the object store to search for the object. If the value is null, the default store is used.

12.2.2. The `deactivate` Method

```
boolean deactivate ()
boolean deactivate (String storeRoot)
```

The `deactivate` is the inverse of activate. It first calls the `save_state` method to build the compacted image of the object, then saves the object in the object store. Objects are only saved if they have been modified since activation. The parameter represents the root name of the object store into which the object should be saved. If the value is null, the default store is used.

12.2.3. The `modified` Method

```
void modified ()
```

The `modified` method must be called before modifying the object in memory. Otherwise, the object will not be saved in the object store by the `deactivate` method.

CHAPTER 13. TOOLS

13.1. INTRODUCTION

This chapter explains how to start and use the tools framework and discusses the available tools.

13.2. STARTING THE TRANSACTION SERVICE TOOLS

To start the JBoss Transaction Service tools, refer to the section below for your operating system.

Windows

Double-click the `Start Tools` link in the JBoss Transaction Service program group in the `Start` menu.

Linux and UNIX-like Operating Systems

At the command line, type *$JBOSS_HOME/*`run-tools.sh`

The Tools window which appears is the launch area for all of the tools shipped with the JBoss Transaction Service. A menu bar is at the top of the Tools window, and contains the following four items:

- Section 13.2.1, "**File** Menu"

- Section 13.2.2, "**Performance** Menu"

- Section 13.2.3, "**Window** Menu"

- Section 13.2.4, "**Help** Menu"

13.2.1. File Menu

Open JMX Browser

Displays the JMX browser window

Open Object Store Browser

Displays the Object Store browser window

Settings

Displays the settings dialog used to configure JBoss Transaction Service tools

Exit

Exits the JBoss Transaction Service Tools application, discarding any unsaved changes.

13.2.2. Performance Menu

Open

Opens a performance window. Refer to Section 13.3, "Using the Performance Tool" for more information on the performance tool.

Close All

Closes all open performance windows.

13.2.3. Window Menu

Cascade Windows

Arranges the windows in a diagonal stack, leaving all the title bars visible.

List of Individual Windows

For each window currently visible, an extra menu option is shown When selected, it focuses the associated window.

13.2.4. Help Menu

About

Displays information about the product, version, authors, and licensing.

13.3. USING THE PERFORMANCE TOOL

The performance tool displays performance information about the transaction service. This information is gathered using the Performance JMX bean, so the transaction service needs to be integrated into an Application Server to give any performance information.

The information is displayed using a multi-series graph. To view this graph, open a performance window by selecting **PerformanceOpen**.

Items Displayed in the Performance Tool Graph

- Number of transactions.

- Number of committed transactions.

- Number of aborted transactions.

- Number of nested transactions.

- Number of heuristics raised.

To toggle these series on and off, select the menu option from the `Series` menu.

If a series is active, it appears in the legend at the bottom of the graph, along with its color in the graph.

The Y-axis represents the number of transactions and the X-axis represents time.

At any point the sampling of data can be stopped and restarted using the **Sampling** menu and the data currently visible in the graph can be saved to a Comma Separate Values (CSV) file for importing the data into a spreadsheet application using the **Save to .csv** menu option from the **Data** menu.

13.4. USING THE JMX BROWSER

To open the JMX browser window, choose **FileOpen JMX Browser** option.

The window consists of the MBean panel and the Details panel. The MBean panel displays the MBeans exposed by the MBean server, grouped by domain name. The Details panel displays information about the currently selected MBean. To select an MBean, click it with the mouse.

Information Displayed in the Details Panel

- The total number of MBeans registered on this server.

- The number of constructors exposed by this MBean.

- The number of attributes exposed by this MBean.

- The number of operations exposed by this MBean.

- The number of notifications exposed by this MBean.

- A brief description of the MBean.

A link to the **View** menu enables display of the attributes and operations exposed by the MBean. You can view readable attributes, alter writeable attributes and invoke operations.

13.4.1. Using Attributes and Operations

Clicking the **View** link displays the View JMX Attributes and Operations window. Use this window to view all readable attributes exposed by the selected MBean, and alter writeable attributes. To alter an attribute's value, double-click the current value and enter the new value. If the `...` button is enabled, click it to enable an advanced editor. If the attribute is a JMX object name, clicking this button displays the JMX attributes and operations for that object.

At any point, you can click the `Refresh` button to refresh the attribute values. If an exception occurs while retrieving the value of an attribute, the exception is displayed in place of the attributes value.

You can also invoke operations upon an MBean, by selecting them from the list of operations, which is displayed below the attributes list, and clicking the `Invoke` button. If the operation requires parameters a further window will be displayed, from this window you must specify values for each of the parameters required. You specify parameter values in the same way as you specify JMX attribute values. Once you have specified a value for each of the parameters click the `Invoke` button to perform the invocation. After the invocation has completed, its return value is displayed.

13.4.2. Using the Object Store Browser

To open the Object Store browser window, click the **FileOpen Object Store Browser** option.

The object store browser window is split into four sections:

Object Store Roots

A drop-down list of the currently-available object store roots. Selecting an option from the list repopulates the hierarchy view with the contents of the selected root.

Object Store Hierarchy

A tree showing the current object store hierarchy. Selecting a node from this tree displays the objects stored in that location.

Objects

A list of icons representing the objects stored in the selected location.

Object Details

Information about the currently selected object, only displayed if the object's type is known to the state viewer repository. Refer to Section 13.4.3.1, "Writing an OSV" to ensure that your object will be displayed properly.

13.4.3. Object State Viewers (OSV)

When an object is selected in the `Objects` pane of the main window, the registered *Object State Viewer (OSV)* for the object's type is invoked. An OSV makes information about the selected object available via the user interface. An OSV for Atomic Actions (transactions) is distributed with the standard tools. It displays information on the Abstract Records in its lists of methods, such as heuristic, failed, read-only, and others. You can write your own OSVs to display information about object types you have defined.

13.4.3.1. Writing an OSV

You can write an OSV plug-in so that the Object Store browser can show the state of user-defined abstract records. An OSV plug-in is a class which implements the `com.arjuna.ats.tools.objectstorebrowser.stateviewers.StateViewerInterface` interface.

It must be packaged in a JAR within the **plugins/** directory. The example at Example 13.1, "**AbstractRecord** Class" creates an OSV plugin for the **AbstractRecord** class.

Example 13.1. AbstractRecord Class

```java
public class SimpleRecord extends AbstractRecord
{
private int _value = 0;

.....

public void increase()
{
_value++;
}

public int get()
{
return _value;
}

public String type()
{
return "/StateManager/AbstractRecord/SimpleRecord";
}

public boolean restore_state(InputObjectState os, int i)
{
boolean returnValue = true;
```

```
    try
    {
    _value = os.unpackInt();
    }
    catch (java.io.IOException e)
    {
    returnValue = false;
    }

    return returnValue;
    }

    public boolean save_state(OutputObjectState os, int i)
    {
    boolean returnValue = true;

    try
    {
    os.packInt(_value);
    }
    catch (java.io.IOException e)
    {
    returnValue = false;
    }

    return returnValue;
    }
    }
```

The goal is to show the current value of the abstract record when it is viewed in the object store browser. You can read the state into an instance of your abstract record, and call the **getValue()** method to accomplish this easily.

```
    public class SimpleRecordOSVPlugin implements StateViewerInterface
    {
    /**
    * A uid node of the type this viewer is registered against has been
    expanded.
    * @param os
    * @param type
    * @param manipulator
    * @param node
    * @throws ObjectStoreException
    */
    public void uidNodeExpanded(ObjectStore os,
    String type,
    ObjectStoreBrowserTreeManipulationInterface
    manipulator,
    UidNode node,
    StatePanel infoPanel)
    throws ObjectStoreException
    {
    // Do nothing
```

```
        }

        /**
        * An entry has been selected of the type this viewer is registered
        against.
        *
        * @param os
        * @param type
        * @param uid
        * @param entry
        * @param statePanel
        * @throws ObjectStoreException
        */
        public void entrySelected(ObjectStore os,
        String type,
        Uid uid,
        ObjectStoreViewEntry entry,
        StatePanel statePanel)
        throws ObjectStoreException
        {
        SimpleRecord rec = new SimpleRecord();

        if ( rec.restore_state( os.read_committed(uid, type),
    ObjectType.ANDPERSISTENT ) )
        {
        statePanel.setData( "Value", rec.getValue() );
        }
        }

        /**
        * Get the type this state viewer is intended to be registered against.
        * @return
        */
        public String getType()
        {
        return "/StateManager/AbstractRecord/SimpleRecord";
        }
        }
```

The **uidNodeExpanded** method is invoked when a *Unique Identification (UID)* representing the given type is expanded in the object store hierarchy tree. This abstract record is not visible in the object store directly. It is only viewable via one of the lists in an atomic action. The **entrySelected** method is invoked when an entry is selected from the Object view, which represents an object with the given type. In both methods, the StatePanel is used to display information regarding the state of the object. The StatePanel includes the methods listed in StatePanel Methods to assist in display this information.

StatePanel Methods

setInfo(String *info*)

Shows general information.

setData(String *name*, String *value*)

Puts information into the table which is displayed by the object store browser tool.

enableDetailsButton(DetailsButtonListener *listener*)

> Enable the **Details** button. The listener interface allows a plug-in to be informed when the button is pressed. You, as the developer, control how to display this information.

In this example, the state is read from the object store and the value returned by the **getValue()** method is used to put an entry into the state panel table. The **getType()** method returns the type of this plug-in for registration.

To add this plug-in to the object store browser, package it into a JAR file with a name that is prefixed with *osbv-*. The JAR file must contain certain information within the manifest file, to inform the object store browser which classes are plug-ins. Refer to Example 13.2, "Packing the Plug-In with Ant" to do this using an Apache Ant script.

Example 13.2. Packing the Plug-In with Ant

```
<jar jarfile="osbv-simplerecord.jar">
<fileset dir="build" includes="*.class"/>
<manifest>
<section name="arjuna-tools-objectstorebrowser">
<attribute name="plugin-classname-1" value=" SimpleRecordOSVPlugin
"/>
</section>
</manifest>
</jar>
```

After you have created the JAR with the correct information in the manifest file, place it in the **bin/tools/plugins** directory.

CHAPTER 14. CONSTRUCTING AN APPLICATION USING TRANSACTIONAL OBJECTS FOR JAVA

14.1. APPLICATION CONSTRUCTION

Developing a JBoss Transaction Service application involves two distinct phases. First, the class developer writes new classes which need to be persistent, recoverable, or concurrency-controlled. Then, the application developer uses the classes you've created in your application. These developers may be the same person, but the two different roles imply different concerns. The class developer needs to focus on developing appropriate **save_state** and **restore_state** methods, as well as setting appropriate locks in operations and invoking the appropriate JBoss Transaction Service class constructors. The application developer's concern is defining the general structure of the application, particularly with regard to the use of atomic actions, or transactions.

This chapter outlines a simple application, a simple FIFO Queue class for integer values. The Queue uses a double linked list structure, and is implemented as a single object. The example is used throughout the remainder of this manual, to illustrate the various mechanisms provided by JBoss Transaction Service. Although the example is simplistic, it shows all possible modifications to JBoss Transaction Service without requiring in-depth knowledge of the application code.

Examples in this chapter assume that the application is not distributed. In a distributed application, context information must be propagated either implicitly or explicitly.

14.1.1. Queue description

The queue is a traditional FIFO queue, where elements are added to the front and removed from the back. The operations provided by the queue class allow the values to be placed into the queue (*enqueue*) and to be removed from it (*dequeue*), as well as the ability to change or inspect the values of elements in the queue. In this example implementation, an array is used to represent the queue. A limit of **QUEUE_SIZE** elements has been imposed for this example.

Example 14.1. Java Interface Definition of the Que Class

```java
public class TransactionalQueue extends LockManager
{
public TransactionalQueue (Uid uid);
public TransactionalQueue ();
public void finalize ();

public void enqueue (int v) throws OverFlow, UnderFlow,
QueueError, Conflict;
public int dequeue  () throws OverFlow, UnderFlow,
QueueError, Conflict;

public int queueSize ();
public int inspectValue (int i) throws OverFlow,
UnderFlow, QueueError, Conflict;
public void setValue (int i, int v) throws OverFlow,
UnderFlow, QueueError, Conflict;

public boolean save_state (OutputObjectState os, int ObjectType);
public boolean restore_state (InputObjectState os, int ObjectType);
public String type ();
```

```
    public static final int QUEUE_SIZE = 40; // maximum size of the
queue

    private int[QUEUE_SIZE] elements;
    private int numberOfElements;
    };
```

14.1.2. Constructors and deconstructors

Example 14.2. Using an Existing Persistent Object

To use an existing persistent object, you need to use a special constructor that is required to take the Uid of the persistent object.

```
    public TransactionalQueue (Uid u)
    {
    super(u);

    numberOfElements = 0;
    }
```

Example 14.3. Creating a New Persistent Object

```
    public TransactionalQueue ()
    {
    super(ObjectType.ANDPERSISTENT);

    numberOfElements = 0;

    try
    {
    AtomicAction A = new AtomicAction();

    A.begin(0); // Try to start atomic action

    // Try to set lock

    if (setlock(new Lock(LockMode.WRITE), 0) == LockResult.GRANTED)
    {
    A.commit(true); // Commit
    }
    else  // Lock refused so abort the atomic action
    A.rollback();
    }
    catch (Exception e)
    {
    System.err.println("Object construction error: "+e);
    System.exit(1);
    }
    }
```

To use an atomic action within the constructor for a new object, follow the guidelines outlined earlier, which ensure that the state of the object is written to the object store when the appropriate top-level atomic action commits. To use atomic actions in a constructor, you need to first declare the action and invoke its **begin** method. Then, the operation must set an appropriate lock on the object. Afterward, the main body of the constructor is executed. If successful, the atomic action is committed. Otherwise, it is aborted.

The destructor of the queue class only needs to call the `terminate` method of the **LockManager** method.

Example 14.4. Destructor of the Queue Class

```
public void finalize ()
{
super.terminate();
}
```

14.1.3. The `save_state`, `restore_state`, and `type` Methods

Example 14.5. The save_state Method

```
public boolean save_state (OutputObjectState os, int ObjectType)
{
if (!super.save_state(os, ObjectType))
return false;

try
{
os.packInt(numberOfElements);

if (numberOfElements > 0)
{
for (int i = 0; i < numberOfElements; i++)
os.packInt(elements[i]);
}

return true;
}
catch (IOException e)
{
return false;
}
}
```

Example 14.6. The restore_state Method

```
public boolean restore_state (InputObjectState os, int ObjectType)
{
if (!super.restore_state(os, ObjectType))
return false;
```

```
try
{
numberOfElements = os.unpackInt();

if (numberOfElements > 0)
{
for (int i = 0; i < numberOfElements; i++)
elements[i] = os.unpackInt();
}

return true;
}
catch (IOException e)
{
return false;
}
}
```

Example 14.7. The type Method

Because the Queue class is derived from the **LockManager** class, the operation type should return a transactional queue.

```
public String type ()
{
return "/StateManager/LockManager/TransactionalQueue";
}
```

14.1.4. enqueue/dequeue operations

If the operations of the queue class will be atomic actions, then the **enqueue** operation in Example 14.8, "The **enqueue** Method" is appropriate as a guideline. The **dequeue** would have a similar structure, but is not included as an example.

Example 14.8. The enqueue Method

```
public void enqueue (int v) throws OverFlow, UnderFlow, QueueError
{
AtomicAction A = new AtomicAction();
boolean res = false;

try
{
A.begin(0);

if (setlock(new Lock(LockMode.WRITE), 0) == LockResult.GRANTED)
{
if (numberOfElements < QUEUE_SIZE)
{
elements[numberOfElements] = v;
numberOfElements++;
res = true;
```

```
        }
        else
        {
        A.rollback();
        throw new UnderFlow();
        }
        }

        if (res)
        A.commit(true);
        else
        {
        A.rollback();
        throw new Conflict();
        }
        }
        catch (Exception e1)
        {
        throw new QueueError();
        }
        }
```

14.1.5. The `queueSize` Method

```
    public int queueSize () throws QueueError, Conflict
    {
    AtomicAction A = new AtomicAction();
    int size = -1;

    try
    {
    A.begin(0);

    if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
    size = numberOfElements;

    if (size != -1)
    A.commit(true);
    else
    {
    A.rollback();

    throw new Conflict();
    }
    }
    catch (Exception e1)
    {
    throw new QueueError();
    }

    return size;
    }
```

14.1.6. The `inspectValue` and `setValue` Methods

> **NOTE**
>
> The implementation of the **setValue** is not shown, but it can be inferred from the **inspectValue** method which is shown.

```java
public int inspectValue (int index) throws UnderFlow,
OverFlow, Conflict, QueueError
{
AtomicAction A = new AtomicAction();
boolean res = false;
int val = -1;

try
{
A.begin();

if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
{
if (index < 0)
{
A.rollback();
throw new UnderFlow();
}
else
{
// array is 0 - numberOfElements -1

if (index > numberOfElements -1)
{
A.rollback();
throw new OverFlow();
}
else
{
val = elements[index];
res = true;
}
}
}

if (res)
A.commit(true);
else
{
A.rollback();
throw new Conflict();
}
}
catch (Exception e1)
{
throw new QueueError();
}
```

```
    return val;
  }
```

14.1.7. The Client

The example code for the client only includes a representative portion, rather than the full code. Before invoking operations on the object, the client needs to bind to it. If the client is run locally, it only needs to create an instance of the object.

Example 14.9. Creating an Instance of the `TransactionalQueue` Object

```
public static void main (String[] args)
{
TransactionalQueue myQueue = new TransactionalQueue();
```

Before invoking one of the queue's operations, the client starts a transaction, using the **queueSize** method.

Example 14.10. The queueSize Method

```
AtomicAction A = new AtomicAction();
int size = 0;

try
{
A.begin(0);
s
try
{
size = queue.queueSize();
}
catch (Exception e)
{
}

if (size >= 0)
{
A.commit(true);

System.out.println("Size of queue: "+size);
}
else
A.rollback();
}
catch (Exception e)
{
System.err.println("Caught unexpected exception!");
}
```

14.1.8. Notes

Because the queue object is persistent, the state of the object will survive any failures of the node on which it is located. The preserved state will be the one produced by the last top-level committed atomic action performed on the object. If the application needs to perform two **enqueue** operations atomically, you can nest the **enqueue** operations inside another enclosing atomic action. In addition, concurrent operations on a persistent object are serialized, preventing inconsistencies in the state of the object. Be aware that since the elements of the queue objects are not individually concurrency controlled, certain combinations of concurrent operation invocations are executed serially, when logically they could be executed concurrently. This happens when modifying the states of two different elements in the queue.

CHAPTER 15. CONFIGURATION OPTIONS

15.1. OPTIONS

Table 15.1, "Configuration Options" shows the configuration options, with possible values. More details about each option can be found in the relevant sections of this document.

Table 15.1. Configuration Options

Name	Values	Description
com.arjuna.ats.arjuna.objectstore.storeSync	ON/OFF	Enables or disables synchronization of the object store. Use with caution.
com.arjuna.ats.arjuna.objectstore.storeType	ShadowStore ShadowNoFileLockStore JDBCStore HashedStore	Specifies the type of object store implementation to use.
com.arjuna.ats.arjuna.objectstore.hashedDirectories	Any integer	Sets the number of directories to hash object states over for the **HashedStore** object store implementation.
com.arjuna.ats.txoj.lockstore.lockStoreType	BasicLockStore BasicPersistentLockStore	Specifies the type of lock store implementation to use.
com.arjuna.ats.txoj.lockstore.lockStoreDir	Windows: .\LockStore Unix: ./LockStore	Specifies the location of the lock store.
com.arjuna.ats.arjuna.objectstore.objectStoreDir	Any location the application can write to.	Specifies the location of the object store.
com.arjuna.ats.arjuna.objectstore.localOSRoot	defaultStore	Specifies the name of the object store root.

Name	Values	Description
com.arjuna.ats.arjuna.coordinator.actionStore	ActionStore HashedActionStore JDBCActionStore	The transaction log implementation to use.
com.arjuna.ats.arjuna.coordinator.asyncCommit	YES/NO	Turns on or off asynchronous `commit`. Off by default.

Table 15.2. Configuration Options (part 2)

Name	Values	Description
com.arjuna.ats.arjuna.coordinator.asyncPrepare	YES/NO	Turns on or off asynchronous `prepare`. Off by default.
com.arjuna.ats.arjuna.objectstore.transactionSync	ON/OFF	Turns synchronization of the object store on or off. Use with caution.
com.arjuna.ats.arjuna.objectstore.jdbcUserDbAccess	JDBCAccess class name	Specifies the JDBCAccess implementation to use for user-level object stores.
com.arjuna.ats.arjuna.objectstore.jdbcTxDbAccess	JDBCAccess class name	Specifies the JDBCAccess implementation to use for transaction object stores.
com.arjuna.ats.arjuna.coordinator.commitOnePhase	YES/NO	Enables or disables the one-phase `commit` optimization.

Name	Values	Description
com.arjuna.ats.arjuna.coordinator.readonlyOptimisation	YES/NO	Enables or disables read-only optimization for the second phase `abort`.
com.arjuna.ats.arjuna.coordinator.enableStatistics	YES/NO	Starts or stops collecting transaction statistic information.
com.arjuna.ats.arjuna.coordinator.startDisabled	YES/NO	Start with the transaction system enabled or disabled. Toggle via the class mentioned in the footnote.[a]
com.arjuna.ats.arjuna.coordinator.defaultTimeout	Any integer	Timeout, in milliseconds

[a] Toggle via the `com.arjuna.ats.arjuna.coordinator.TxControl` class.

APPENDIX A. OBJECT STORE IMPLEMENTATIONS

A.1. THE OBJECTSTORE

This chapter covers the various JBoss Transaction Service object store implementations and provides guidelines for creating new implementations and using them in an application.

JBoss Transaction Service includes several different implementations of a basic object store. Each implementation is optimized for a particular purpose. All of the implementations are derived from the ObjectStore interface, which defines the minimum operations which is needed for an object store implementation to be used by JBoss Transaction Service. You can override the default object store implementation at runtime by setting the com.arjuna.ats.arjuna.objectstore.objectStoreType property to one of the types described below in Example A.1, "Object Store Type".

Example A.1. Object Store Type

```
/*
 * This is the base class from which all object store types are derived.
 * Note that because object store instances are stateless, to improve
 * efficiency we try to only create one instance of each type per
process.
 * Therefore, the create and destroy methods are used instead of new
 * and delete. If an object store is accessed via create it *must* be
 * deleted using destroy. Of course it is still possible to make use of
 * new and delete directly and to create instances on the stack.
 */

public class ObjectStore
{
public static final int OS_COMMITTED;
public static final int OS_COMMITTED_HIDDEN;
public static final int OS_HIDDEN;
public static final int OS_INVISIBLE;
public static final int OS_ORIGINAL;
public static final int OS_SHADOW;
public static final int OS_UNCOMMITTED;
public static final int OS_UNCOMMITTED_HIDDEN;
public static final int OS_UNKNOWN;
public ObjectStore (ClassName type);
public ObjectStore (ClassName type, String osRoot);
public ObjectStore (String osRoot);
public synchronized boolean allObjUids (String s, InputObjectState
buff)
 throws ObjectStoreException;
public synchronized boolean allObjUids (String s, InputObjectState
buff,
 int m) throws ObjectStoreException;

public synchronized boolean allTypes (InputObjectState buff)
 throws ObjectStoreException;
public synchronized int currentState(Uid u, String tn)
 throws ObjectStoreException;
public synchronized boolean commit_state (Uid u, String tn)
 throws ObjectStoreException;
public synchronized boolean hide_state (Uid u, String tn)
```

```
    throws ObjectStoreException;
    public synchronized boolean reveal_state (Uid u, String tn)
    throws ObjectStoreException;
    public synchronized InputObjectState read_committed (Uid u, String tn)
    throws ObjectStoreException;
    public synchronized InputObjectState read_uncommitted (Uid u, String
tn)
    throws ObjectStoreException;
    public synchronized boolean remove_committed (Uid u, String tn)
    throws ObjectStoreException;
    public synchronized boolean remove_uncommitted (Uid u, String tn)
    throws ObjectStoreException;
    public synchronized boolean write_committed (Uid u, String tn,
    OutputObjectState buff)
    throws ObjectStoreException;
    public synchronized boolean write_uncommitted (Uid u, String tn,
    OutputObjectState buff)
    throws ObjectStoreException;
    public static void printState (PrintStream strm, int res);
    };
```

You do not usually need to interact with any of the object store implementations directly, except for creating them if you are not using the default store type. All stores manipulate instances of the class **ObjectState**, which are named using a type derived via the object's **type()** operation, and a Uid. Object states in the store are usually in one of two distinct states **OS_COMMITTED** or **OS_UNCOMMITTED**. An object state starts in the **OS_COMMITTED** state, but when modified under the control of an atomic action, a new second object state may be written that is in the **OS_UNCOMMITTED** state. If the action commits, this second object state replaces the original and becomes **OS_COMMITTED**. If the action aborts, this second object state is discarded. All of the implementations provided with this release use shadow copies to handle these state transitions. However, you are allowed to implement them in a different way. Object states may become hidden and inaccessible under the control of the crash recovery system.

The **allTypes** and **allObjUids** methods provide the ability to browse the contents of a store. The **allTypes** method returns an InputObjectState containing all of the type names of all objects in a store, terminated by a null name. The **allObjUids** method returns an InputObjectState that contains all of the Uids of all objects of a given type terminated by the special **Uid.nullUid()** type.

A.2. PERSISTENT OBJECT STORES

This section briefly describes the characteristics and optimizations of each of the supplied implementations of the persistent object store. Persistent object states are mapped onto the structure of the file system supported by the host operating system.

Common Functionality

In addition to the features mentioned earlier, all of the supplied persistent object stores obey the following rules:

- Each object state is stored in its own file, which is named using the **Uid** of the object.

- The type of an object, provided by the **type()** operation, determines the directory where the object is placed.

- All of the stores have a common root directory which is determined by the JBoss Transaction Service configuration. This directory name is automatically prepended to any store-specific root information.

- All stores can also use a localized root directory that is automatically prepended to the type of the object to determine the ultimate directory name. The localized root name is specified when the store is created. The default localized root name is `defaultStore`.

Table A.1. Example Object Store Information

Item	Example Value
ObjectStore root Directory from configure	`/JBossTS/ObjectStore/`
ObjectStore Type 1	`FragmentedStore/`
Default root	`defaultStore/`
StateManager	`StateManager`
LockManager	`LockManager/`
User Types	
Localised root 2	`myStore/`
StateManager	`StateManager/`
ObjectStore Type2	`ActionStore/`
Default root	`defaultStore/`

A.2.1. The Shadowing Store

The shadowing store is the original version of the object store as provided in prior releases. It is implemented by the `ShadowingStore` class. It is simple but slow, using pairs of files to represent objects: the shadow version and the committed version. Files are opened, locked, operated upon, unlocked, and closed during every interaction with the object store. This can take more resources than strictly necessary just to open, close, and rename files.

The type of this object store is ShadowingStore.

A.2.2. No file-level locking

Since transactional objects are concurrency-controlled through the `LockManager` method, no additional locking is needed at the file level. Therefore, the default object store implementation for JBoss Transaction Service, `ShadowNoFileLockStore`, relies upon user-level locking, enabling it to provide better performance than the `ShadowingStore` implementation.

The type of this object store is ShadowNoFileLockStore.

A.2.3. The Hashed Store

The **HashedStore** implementation uses the same structure for object states as the shadowing store, but uses an alternate directory structure that is designed to store large numbers of objects of the same type. Objects are scattered throughout a set of directories by means of a hashing function which uses the object's Uid. By default, 255 sub-directories are used, but you can override this by setting the **HASHED_DIRECTORIES** environment variable.

The type of this object store is HashedStore.

A.2.4. The JDBC Store

The **JDBCStore** implementation stores persistent object states in a JDBC database. Nested transaction support is available when the **JBDCStore** is used in conjunction with the Transactional Objects for Java API. All object states are stored as *Binary Large Objects (BLOBs)* within a single table. Object state size is limited to 64k. If you try to store an object state which exceeds this limit, an exception is thrown and the state is not stored. The transaction is forced to roll back.

When using the JDBC object store, the application needs to provide an implementation of the **JDBCAccess** interface, located in the com.arjuna.ats.arjuna.objectstore package. See the Example A.2, "JDBCAccess Implementation Example" .

> **Example A.2. JDBCAccess Implementation Example**
>
> ```
> public interface JDBCAccess
> {
> public Connection getConnection () throws SQLException;
> public void putConnection (Connection conn) throws SQLException;
> public void initialise (ObjectName objName);
> }
> ```

The implementation of the **JDBCAccess** class provides the **Connection** used by the JDBC ObjectStore to save and restore object states. Refer to **JDBCAccess Connection** Methods for details.

JDBCAccess Connection Methods

getConnection

Returns the Connection to use. This method will be called whenever a connection is required and the implementation should use whatever policy is necessary for determining what connection to return. This method need not return the same Connection instance more than once.

putConnection

Returns one of the Connections acquired from getConnection. Connections are returned if any errors occur when using them.

initialise

Passes additional arbitrary information to the implementation.

The JDBC object store initially requests the number of **Connections** defined in the com.arjuna.ats.arjuna.objectstore.jdbcPoolSizeInitial property, and uses no more than defined in the

com.arjuna.ats.arjuna.objectstore.jdbcPoolSizeMaximum property.

The implementation of the **JDBCAccess** interface to use should be set in the com.arjuna.ats.arjuna.objectstore.jdbcUserDbAccess property variable.

The type of this object store is JDBCStore.

A JDBC object store can manage the transaction log. The transaction log implementation should be set to JDBCActionStore, and the **JDBCAccess** method should be provided via the com.arjuna.ats.arjuna.objectstore.jdbcTxDbAccess property. The default table name is **JBossTSTxTable**.
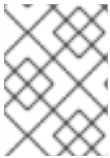
> **NOTE**
>
> You can use the same **JDBCAccess** implementation for both the user object store and the transaction log.

A.2.5. The Cached Store

The cached store uses the hashed object store, but does not read or write states to the persistent backing store immediately. It maintains the states in a volatile memory cache, flushing the cache periodically or when it is full. The failure semantics associated with this object store are different from the semantics used with the normal persistent object stores, because data about states could be lost in the event of a failure.

The type of this object store is CachedStore.

Configuration Options for the Cached Store

com.arjuna.ats.internal.arjuna.objectstore.cacheStore.hash

Sets the number of internal stores to hash the states over. The default value is **128**.

com.arjuna.ats.internal.arjuna.objectstore.cacheStore.size

The maximum size the cache can reach before a flush is triggered. The default is value is **10240** bytes.

com.arjuna.ats.internal.arjuna.objectstore.cacheStore.removedItems

The maximum number of removed items that the cache can contain before a flush is triggered. By default, calls to remove a state that is in the cache actually only remove the state from the cache, leaving a blank entry. This improves performance. The entries are removed when the cache is flushed. The default value is twice the size of the hash.

com.arjuna.ats.internal.arjuna.objectstore.cacheStore.workItems

The maximum number of items the cache is able to contain before it is flushed. The default value is **100**.

com.arjuna.ats.internal.arjuna.objectstore.cacheStore.scanPeriod

The length of time, in milliseconds, for periodically flushing the cache. The default is **120** seconds.

com.arjuna.ats.internal.arjuna.objectstore.cacheStore.sync

Determines whether flushes of the cache are synchronized to disk. The default is **OFF**, and the other possible value is **ON.**

APPENDIX B. CLASS DEFINITIONS

B.1. INTRODUCTION

This appendix contains an overview of the most typically-used classes, as a quick reference guide for JBoss Transaction Service. For clarity, only the public and protected interfaces of the classes are given.

B.2. CLASS LIBRARY

Example B.1. Lock Manager

```
public class LockResult
{
public static final int GRANTED;
public static final int REFUSED;
public static final int RELEASED;
};

public class ConflictType
{
public static final int CONFLICT;
public static final int COMPATIBLE;
public static final int PRESENT;
};

public abstract class LockManager extends StateManager
{
public static final int defaultRetry;
public static final int defaultTimeout;
public static final int waitTotalTimeout;

public final synchronized boolean releaselock (Uid lockUid);
public final synchronized int setlock (Lock toSet);
public final synchronized int setlock (Lock toSet, int retry);
public final synchronized int setlock (Lock toSet, int retry, int
sleepTime);
public void print (PrintStream strm);
public String type ();
public boolean save_state (OutputObjectState os, int ObjectType);
public boolean restore_state (InputObjectState os, int ObjectType);

protected LockManager ();
protected LockManager (int ot);
protected LockManager (int ot, ObjectName attr);
protected LockManager (Uid storeUid);
protected LockManager (Uid storeUid, int ot);
protected LockManager (Uid storeUid, int ot, ObjectName attr);

protected void terminate ();
};
```

Example B.2. StateManager

```
public class ObjectStatus
{
public static final int PASSIVE;
public static final int PASSIVE_NEW;
public static final int ACTIVE;
public static final int ACTIVE_NEW;
};

public class ObjectType
{
public static final int RECOVERABLE;
public static final int ANDPERSISTENT;
public static final int NEITHER;
};

public abstract class StateManager
{
public boolean restore_state (InputObjectState os, int ot);
public boolean save_state (OutputObjectState os, int ot);
public String type ();

public synchronized boolean activate ();
public synchronized boolean activate (String rootName);
public synchronized boolean deactivate ();
public synchronized boolean deactivate (String rootName);
public synchronized boolean deactivate (String rootName, boolean
commit);

public synchronized int status ();
public final Uid get_uid ();
public void destroy ();
public void print (PrintStream strm);

protected void terminate ();

protected StateManager ();
protected StateManager (int ot);
protected StateManager (int ot, ObjectName objName);
protected StateManager (Uid objUid);
protected StateManager (Uid objUid, int ot);
protected StateManager (Uid objUid, int ot, ObjectName objName);
protected synchronized final void modified ();
};
```

Example B.3. Input/OutputObjectState

```
class OutputObjectState extends OutputBuffer
{
public OutputObjectState (Uid newUid, String typeName);

public boolean notempty ();
public int size ();
```

```
public Uid stateUid ();
public String type ();
};
class InputObjectState extends ObjectState
{
public OutputObjectState (Uid newUid, String typeName, byte[] b);

public boolean notempty ();
public int size ();
public Uid stateUid ();
public String type ();
};
```

Example B.4. Input/OutputBuffer

```
public class OutputBuffer
{
public OutputBuffer ();

public final synchronized boolean valid ();
public synchronized byte[] buffer();
public synchronized int length ();

/* pack operations for standard Java types */

public synchronized void packByte (byte b) throws IOException;
public synchronized void packBytes (byte[] b) throws IOException;
public synchronized void packBoolean (boolean b) throws IOException;
public synchronized void packChar (char c) throws IOException;
public synchronized void packShort (short s) throws IOException;
public synchronized void packInt (int i) throws IOException;
public synchronized void packLong (long l) throws IOException;
public synchronized void packFloat (float f) throws IOException;
public synchronized void packDouble (double d) throws IOException;
public synchronized void packString (String s) throws IOException;
};
public class InputBuffer
{
public InputBuffer ();

public final synchronized boolean valid ();
public synchronized byte[] buffer();
public synchronized int length ();

/* unpack operations for standard Java types */

public synchronized byte unpackByte () throws IOException;
public synchronized byte[] unpackBytes () throws IOException;
public synchronized boolean unpackBoolean () throws IOException;
public synchronized char unpackChar () throws IOException;
public synchronized short unpackShort () throws IOException;
public synchronized int unpackInt () throws IOException;
public synchronized long unpackLong () throws IOException;
public synchronized float unpackFloat () throws IOException;
```

```
 public synchronized double unpackDouble () throws IOException;
 public synchronized String unpackString () throws IOException;
 };
```

Example B.5. Uid

```
 public class Uid implements Cloneable
 {
 public Uid ();
 public Uid (Uid copyFrom);
 public Uid (String uidString);
 public Uid (String uidString, boolean errorsOk);
 public synchronized void pack (OutputBuffer packInto) throws
IOException;
 public synchronized void unpack (InputBuffer unpackFrom) throws
IOException;

 public void print (PrintStream strm);
 public String toString ();
 public Object clone () throws CloneNotSupportedException;
 public synchronized void copy (Uid toCopy) throws UidException;
 public boolean equals (Uid u);
 public boolean notEquals (Uid u);
 public boolean lessThan (Uid u);
 public boolean greaterThan (Uid u);

 public synchronized final boolean valid ();
 public static synchronized Uid nullUid ();
 };
```

Example B.6. AtomicAction

```
 public class AtomicAction
 {
 public AtomicAction ();

 public void begin () throws SystemException,
SubtransactionsUnavailable,
 NoTransaction;
 public void commit (boolean report_heuristics) throws SystemException,
 NoTransaction, HeuristicMixed,
 HeuristicHazard,TransactionRolledBack;
 public void rollback () throws SystemException, NoTransaction;
 public Control control () throws SystemException, NoTransaction;
 public Status get_status () throws SystemException;
 /* Allow action commit to be supressed */
 public void rollbackOnly () throws SystemException, NoTransaction;

 public void registerResource (Resource r) throws SystemException,
Inactive;
 public void registerSubtransactionAwareResource
(SubtransactionAwareResource sr)
```

```
  throws SystemException, NotSubtransaction;
 public void registerSynchronization (Synchronization s) throws
SystemException,
 Inactive;
 };
```

PART III. XTS DEVELOPMENT

This section gives guidance for using the JBoss implementation of the XML Transactions (XTS) API to add transactional support for your Web Services applications, and interact with third-party applications across a distributed web environment, in a reliable and data-safe way.

CHAPTER 16. INTRODUCTION

The *XML Transaction Service (XTS)* component of JBoss Transaction Service supports the coordination of private and public Web Services in a business transaction. Therefore, to understand XTS, you must be familiar with Web Services, and also understand something about transactions. This chapter introduces XTS and provides a brief overview of the technologies that form the Web Services standard. Additionally, this chapter explores some of the fundamentals of transactioning technology and how it can be applied to Web Services. Much of the content presented in this chapter is detailed throughout this guide. However, only overview information about Web Services is provided. If you are new to creating Web services, please see consult your Web Services platform documentation.

JBoss Transaction Service provides the XTS component as a transaction solution for Web Services. Using XTS, business partners can coordinate complex business transactions in a controlled and reliable manner. The XTS API supports a transactional coordination model based on the *WS-Coordination*, *WS-Atomic Transaction*, and *WS-Business Activity* specifications.

Protocols Included in XTS

- WS-Coordination (WS-C) is a generic coordination framework developed by IBM, Microsoft and BEA.

- WS-Atomic Transaction (WS-AT) and WS-Business Activity (WS-BA) together comprise the WS-Transaction (WS-T) transaction protocols that utilize this framework.

JBoss Transaction Service implements versions 1.0, 1.1, and 1.2 of these three specifications. Version specifications are available from http://www.oasis-open.org/specs/.
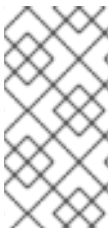
> **NOTE**
>
> The 1.0, 1.1, and 1.2 specifications only differ in a small number of details. The rest of this document employs version 1.1 of these specifications when providing explanations and example code. On the few occasions where the modifications required to adapt these to the 1.1 specifications are not obvious, an explanatory note is provided.

Web Services are modular, reusable software components that are created by exposing business functionality through a Web service interface. Web Services communicate directly with other Web Services using standards-based technologies such as SOAP and HTTP. These standards-based communication technologies enable customers, suppliers, and trading partners to access Web Services, independent of hardware operating system, or programming environment. The result is a vastly improved collaboration environment as compared to today's EDI and *business-to-business (B2B)* solutions, an environment where businesses can expose their current and future business applications as Web Services that can be easily discovered and accessed by external partners.

Web Services, by themselves, are not fault-tolerant. In fact, some of the reasons that the Web Services model is an attractive development solution are also the same reasons that service-based applications may have drawbacks.

Properties of Web Services

- Application components that are exposed as Web Services may be owned by third parties, which provides benefits in terms of cost of maintenance, but drawbacks in terms of having exclusive control over their behavior.

- Web Services are usually remotely located, increasing risk of failure due to increased network travel for invocations.

Applications that have high dependability requirements need a method of minimizing the effects of errors that may occur when an application consumes Web Services. One method of safeguarding against such failures is to interact with an application's Web Services within the context of a *transaction*. A transaction is a unit of work which is completed entirely, or in the case of failures is reversed to some agreed consistent state. The goal, in the event of a failure, is normally to appear as if the work had never occurred in the first place. With XTS, transactions can span multiple Web Services, meaning that work performed across multiple enterprises can be managed with transactional support.

16.1. MANAGING SERVICE-BASED PROCESSES

XTS allows you to create transactions that drive complex business processes, spanning multiple Web Services. Current Web Services standards do not address the requirements for a high-level coordination of services. This is because in today's Web Services applications, which use single request/receive interactions, coordination is typically not a problem. However, for applications that engage multiple services among multiple business partners, coordinating and controlling the resulting interactions is essential. This becomes even more apparent when you realize that you generally have little in the way of formal guarantees when interacting with third-party Web Services.

XTS provides the infrastructure for coordinating services during a business process. By organizing processes as transactions, business partners can collaborate on complex business interactions in a reliable manner, insuring the integrity of their data - usually represented by multiple changes to a database – but without the usual overheads and drawbacks of directly exposing traditional transaction-processing engines directly onto the web. An Evening On the Town demonstrates how an application may manage service-based processes as transactions:

An Evening On the Town

The application in question allows a user to plan a social evening. This application is responsible for reserving a table at a restaurant, and reserving tickets to a show. Both activities are paid for using a credit card. In this example, each service represents exposed Web Services provided by different service providers. XTS is used to envelop the interactions between the theater and restaurant services into a single (potentially) long-running business transaction. The business transaction must insure that seats are reserved both at the restaurant and the theater. If one event fails the user has the ability to decline both events, thus returning both services back to their original state. If both events are successful, the user's credit card is charged and both seats are booked. As you may expect, the interaction between the services must be controlled in a reliable manner over a period of time. In addition, management must span several third-party services that are remotely deployed.

Without the backing of a transaction, an undesirable outcome may occur. For example, the user credit card may be charged, even if one or both of the bookings fail.

An Evening On the Town describes the situations where XTS excels at supporting business processes across multiple enterprises. This example is further refined throughout this guide, and appears as a standard demonstrator (including source code) with the XTS distribution.

16.2. SERVLETS

The WS-Coordination, WS-Atomic Transaction, and WS-Business Activity protocols are based on one-way interactions of entities rather than traditional synchronous request/response RPC-style interactions. One group of entities, called transaction participants, invoke operations on other entities, such as the transaction coordinator, in order to return responses to requests. The programming model is based on peer-to-peer relationships, with the result that all services, whether they are participants, coordinators or clients, must have an *active component* that allows them to receive unsolicited messages.

In XTS, the active component is achieved through deployment of JaxWS endpoints. Each XTS endpoint

that is reachable through SOAP/XML is published via JaxWS, without developer intevention. The only requirement is that transactional client applications and transactional web services must reside within a domain capable of hosting JaxWS endpoints, such as an application server. In the case of the Enterprise Application Platform, JBoss Application Server provides this functionality.

> **NOTE**
>
> The XTS 1.0 protocol implementation is based on servlets.

16.3. SOAP

SOAP has emerged as the *de facto* message format for XML-based communication in the Web Services arena. It is a lightweight protocol that allows the user to define the content of a message and to provide hints as to how recipients should process that message.

16.4. WEB SERVICES DESCRIPTION LANGUAGE (WDSL)

Web Services Description Language (WSDL) is an XML-based language used to define Web service interfaces. An application that consumes a Web service parses the service's WSDL document to discover the location of the service, the operations that the service supports, the protocol bindings the service supports (SOAP, HTTP, etc), and how to access them. For each operation, WSDL describes the format that the client must follow.

CHAPTER 17. TRANSACTIONS OVERVIEW

NOTE

This chapter deals with the theory of transactional Web Services. If you are familiar with these principles, consider this chapter a reference.

Transactions have emerged as the dominant paradigm for coordinating interactions between parties in a distributed system, and in particular to manage applications that require concurrent access to shared data. Much of the JBoss Transaction Service Web Service API is based on contemporary transaction APIs whose familiarity will enhance developer productivity and lessen the learning curve. While the following section provides the essential information that you should know before starting to use XTS for building transactional Web Services, it should not be treated as a definitive reference to all transactional technology.

A classic transaction is a unit of work that either completely succeeds, or fails with all partially completed work being undone. When a transaction is committed, all changes made by the associated requests are made durable, normally by committing the results of the work to a database. If a transaction should fail and is rolled back, all changes made by the associated work are undone. Transactions in distributed systems typically require the use of a transaction manager that is responsible for coordinating all of the participants that are part of the transaction.

The main components involved in using and defining transactional Web Services using XTS are illustrated in Figure 17.1, "Components Involved in an XTS Transaction" .
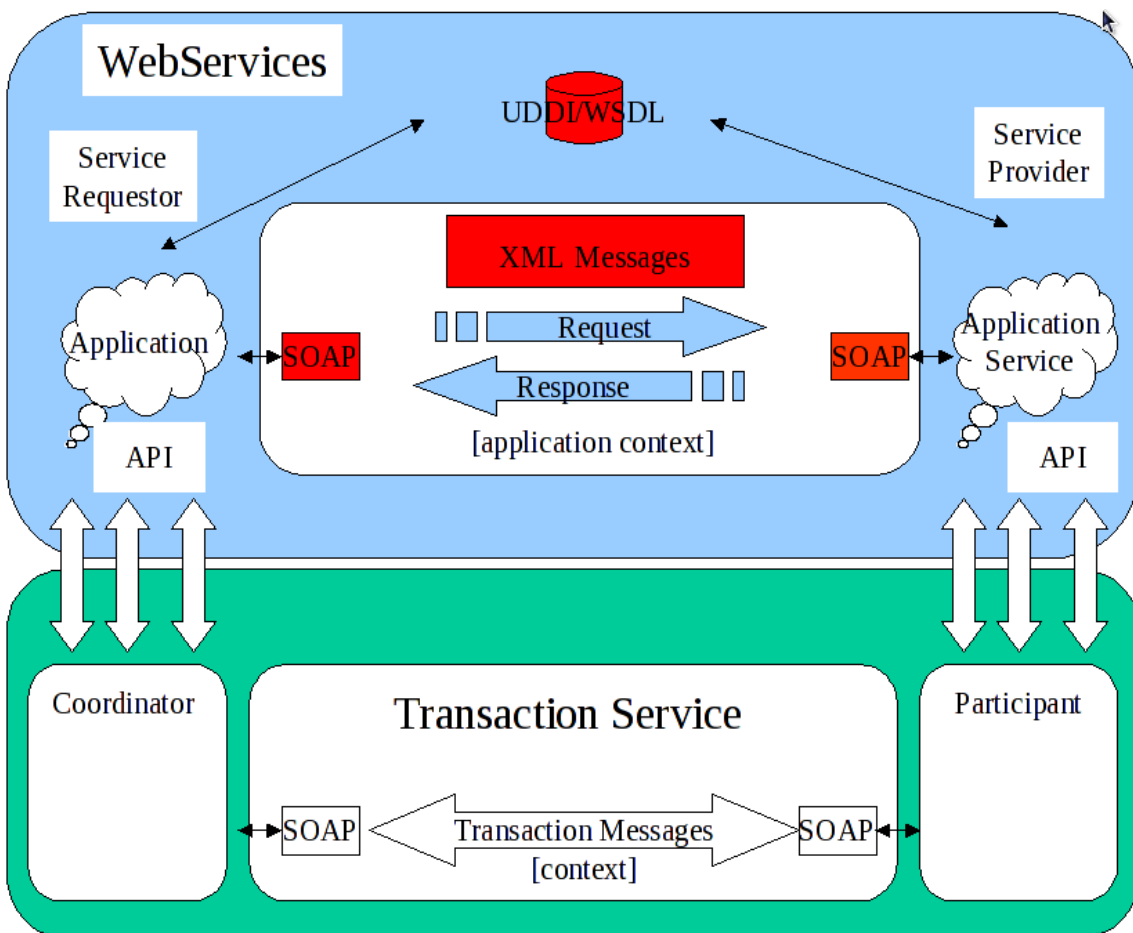

Figure 17.1. Components Involved in an XTS Transaction

17.1. THE COORDINATOR

Every transaction is associated with a coordinator, which is responsible for governing the outcome of the transaction. When a client begins a Web Service transaction it posts a `create` request to a coordination service, which creates the coordinator and returns its details to the client. This service may be located in its own container or may be colocated with the application client or with one of the transactional web services for improved performance. The coordination service is typically responsible for managing many transactions in parallel, so each coordinator is identified by a unique transaction identifier.

The coordinator is responsible for ensuring that the web services invoked by the client arrive at a consistent outcome. When the client asks the coordinator to complete the transaction, the coordinator ensures that each web service is ready to confirm any provisional changes it has made within the scope of the transaction. It then asks them all to make their changes permanent. If any of the web services indicates a problem at the confirmation stage, the coordinator ensures that all web services reject their provisional changes, reverting to the state before the transaction started. The coordinator also reverts all changes if the client asks it to cancel the transaction.

The negotiation between the coordinator and the web services is organized to ensure that all services will make their changes permanent, or all of them will revert to the previous state, even if the coordinator or one of the web services crashes part of the way through the transaction."

17.2. THE TRANSACTION CONTEXT

In order for a transaction to span a number of services, certain information has to be shared between those services, to propagate information about the transaction. This information is known as the *Context*. The coordination service hands a context back to the application client when it begins a transaction. This context is passed as an extra, hidden parameter whenever the client invokes a transactional web service. The XTS implementation saves and propagates this context automatically with only minimal involvement required on the part of the client. However, it is still helpful to understand what information is captured in a context. This information is listed in Contents of a Context.

Contents of a Context

Transaction Identifier

Guarantees global uniqueness for an individual transaction.

Transaction Coordinator Location

The endpoint address participants contact to enroll.
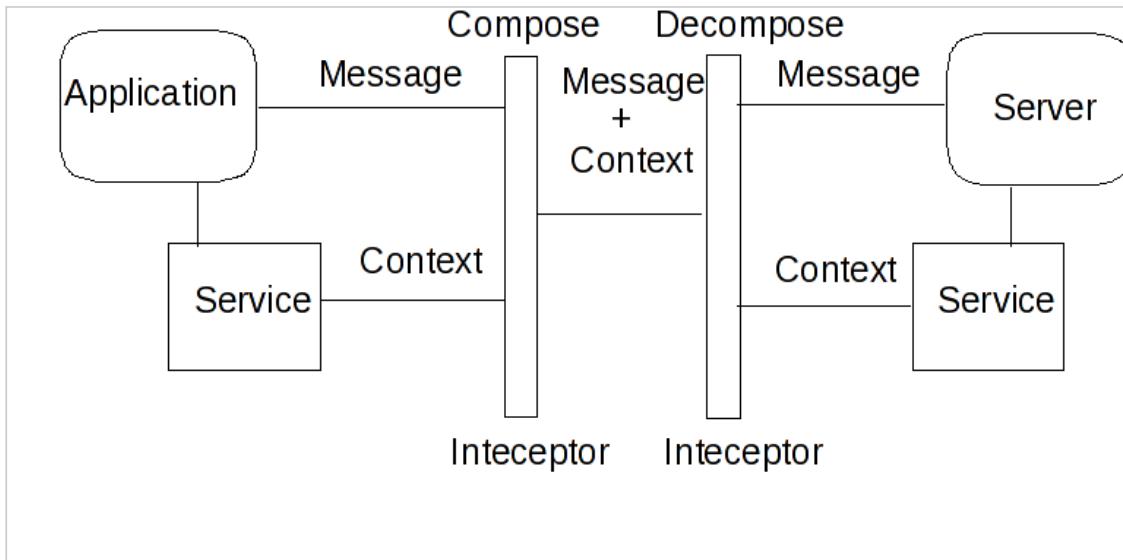
Figure 17.2. Web Services and Context Flow

17.3. PARTICIPANTS

The coordinator cannot know the details of how every transactional service is implemented. In fact this knowledge is not even necessary for it to negotiate a transactional outcome. It treats each service taking part in a transaction as a participant and communicates with it according to some predefined participant coordination models appropriate to the type of transaction. When a web service receives its first service request in some given transaction, it enrolls with the coordinator as a participant, specifying the participant model it wishes to follow. The context contains a URL for the endpoint of the coordination service which handles enrollment requests. So, the term participant merely refers a transactional service enrolled in a specific transaction using a specific participant model.

17.4. ACID TRANSACTIONS

Traditionally, transaction processing systems support *ACID* properties. ACID is an acronym for *A*tomic, *C*onsistent, *I*solated, and *D*urable. A unit of work has traditionally been considered transactional only if the ACID properties are maintained, as describe in ACID Properties.

ACID Properties

Atomicity

The transaction executes completely, or not at all.

Consistency

The effects of the transaction preserve the internal consistency of an underlying data structure.

Isolated

The transaction runs as if it were running alone, with no other transactions running, and is not visible to other transactions.

Durable

The transaction's results are not lost in the event of a failure.

17.5. TWO PHASE COMMIT

The classical two-phase commit approach is the bedrock of JBoss Transaction Service, and more generally of Web Services transactions. Two-phase commit provides coordination of parties that are involved in a transaction. The general flow of a two-phase commit transaction is described in Figure 17.3, "Two-Phase Commit Overview".
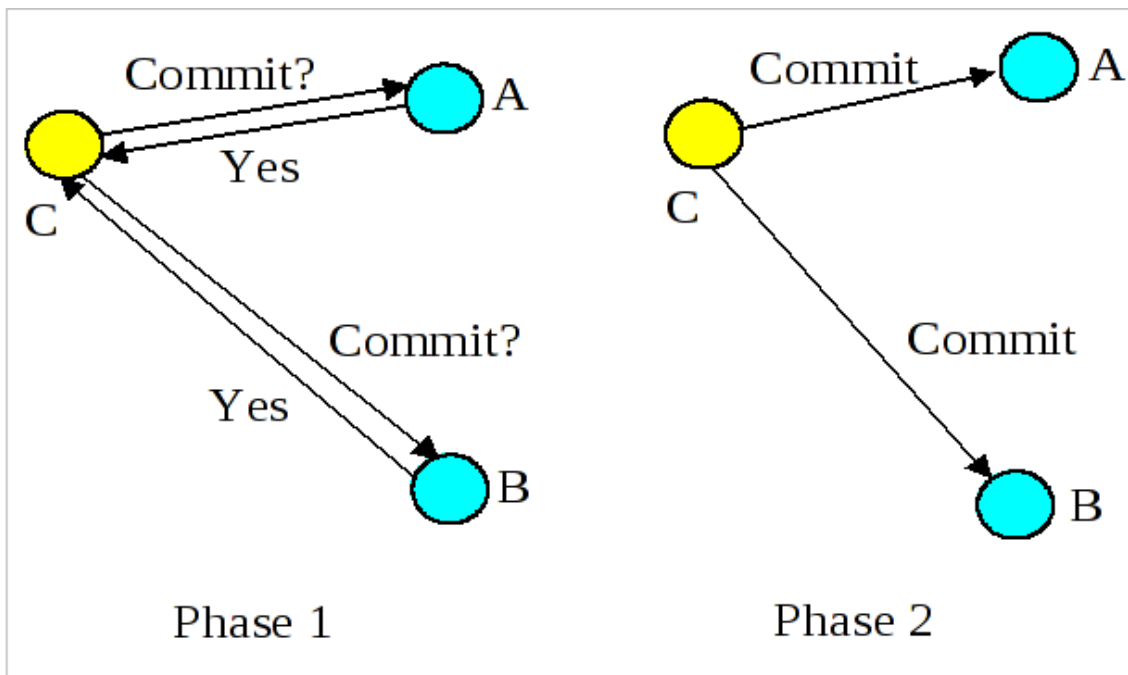
Figure 17.3. Two-Phase Commit Overview
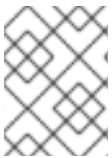
> **NOTE**
>
> During two-phase commit transactions, coordinators and resources keep track of activity in non-volatile data stores so that they can recover in the case of a failure.

17.6. THE SYNCHRONIZATION PROTOCOL

Besides the two-phase commit protocol, traditional transaction processing systems employ an additional protocol, often referred to as the *synchronization protocol*. With the original ACID properties, Durability is important when state changes need to be available despite failures. Applications interact with a persistence store of some kind, such as a database, and this interaction can impose a significant overhead, because disk access is much slower to access than main computer memory.

One solution to the problem disk access time is to cache the state in main memory and only operate on the cache for the duration of a transaction. Unfortunately, this solution needs a way to flush the state back to the persistent store before the transaction terminates, or risk losing the full ACID properties. This is what the synchronization protocol does, with *Synchronization Participants*.

Synchronizations are informed that a transaction is about to commit. At that point, they can flush cached state, which might be used to improve performance of an application, to a durable representation prior to the transaction committing. The synchronizations are then informed about when the transaction completes and its completion state.

Procedure 17.1. The "Four Phase Protocol" Created By Synchronizations

Synchronizations essentially turn the two-phase commit protocol into a four-phase protocol:

1. **Step 1**

   Before the transaction starts the two-phase commit, all registered Synchronizations are informed. Any failure at this point will cause the transaction to roll back.

2. **Steps 2 and 3**

   The coordinator then conducts the normal two-phase commit protocol.

3. **Step 4**

   Once the transaction has terminated, all registered Synchronizations are informed. However, this is a courtesy invocation because any failures at this stage are ignored: the transaction has terminated so there's nothing to affect.

The synchronization protocol does not have the same failure requirements as the traditional two-phase commit protocol. For example, Synchronization participants do not need the ability to recover in the event of failures, because any failure before the two-phase commit protocol completes cause the transaction to roll back, and failures after it completes have no effect on the data which the Synchronization participants are responsible for.

17.7. OPTIMIZATIONS TO THE PROTOCOL

There are several variants to the standard two-phase commit protocol that are worth knowing about, because they can have an impact on performance and failure recovery. Table 17.1, "Variants to the Two-Phase Commit Protocol" gives more information about each one.

Table 17.1. Variants to the Two-Phase Commit Protocol

Variant	Description
Presumed Abort	If a transaction is going to roll back, the coordinator may record this information locally and tell all enlisted participants. Failure to contact a participant has no effect on the transaction outcome. The coordinator is informing participants only as a courtesy. Once all participants have been contacted, the information about the transaction can be removed. If a subsequent request for the status of the transaction occurs, no information will be available and the requester can assume that the transaction has aborted. This optimization has the benefit that no information about participants need be made persistent until the transaction has progressed to the end of the `prepare` phase and decided to commit, since any failure prior to this point is assumed to be an abort of the transaction.
One-Phase	If only a single participant is involved in the transaction, the coordinator does not need to drive it through the `prepare` phase. Thus, the participant is told to commit, and the coordinator does not need to record information about the decision, since the outcome of the transaction is the responsibility of the participant.

Variant	Description
Read-Only	When a participant is asked to prepare, it can indicate to the coordinator that no information or data that it controls has been modified during the transaction. Such a participant does not need to be informed about the outcome of the transaction since the fate of the participant has no affect on the transaction. Therefore, a read-only participant can be omitted from the second phase of the commit protocol.

NOTE

The WS-Atomic Transaction protocol does not support the one-phase commit optimization.

17.8. NON-ATOMIC TRANSACTIONS AND HEURISTIC OUTCOMES

In order to guarantee atomicity, the two-phase commit protocol is *blocking*. As a result of failures, participants may remain blocked for an indefinite period of time, even if failure recovery mechanisms exist. Some applications and participants cannot tolerate this blocking.

To break this blocking nature, participants that are past the `prepare` phase are allowed to make autonomous decisions about whether to commit or rollback. Such a participant must record its decision, so that it can complete the original transaction if it eventually gets a request to do so. If the coordinator eventually informs the participant of the transaction outcome, and it is the same as the choice the participant made, no conflict exists. If the decisions of the participant and coordinator are different, the situation is referred to as a non-atomic outcome, and more specifically as a *heuristic outcome*.

Resolving and reporting heuristic outcomes to the application is usually the domain of complex, manually driven system administration tools, because attempting an automatic resolution requires semantic information about the nature of participants involved in the transactions.

Precisely when a participant makes a heuristic decision depends on the specific implementation. Likewise, the choice the participant makes about whether to commit or to roll back depends upon the implementation, and possibly the application and the environment in which it finds itself. The possible heuristic outcomes are discussed in Table 17.2, "Heuristic Outcomes".

Table 17.2. Heuristic Outcomes

Outcome	Description
Heuristic Rollback	The commit operation failed because some or all of the participants unilaterally rolled back the transaction.

Outcome	Description
Heuristic Commit	An attempted rollback operation failed because all of the participants unilaterally committed. One situation where this might happen is if the coordinator is able to successfully **prepare** the transaction, but then decides to roll it back because its transaction log could not be updated. While the coordinator is making its decision, the participants decides to commit.
Heuristic Mixed	Some participants commit ed, while others were rolled back.
Heuristic Hazard	The disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

Heuristic decisions should be used with care and only in exceptional circumstances, since the decision may possibly differ from that determined by the transaction service. This type of difference can lead to a loss of integrity in the system. Try to avoid needing to perform resolution of heuristics, either by working with services and participants that do not cause heuristics, or by using a transaction service that provides assistance in the resolution process.

17.9. INTERPOSITION

Interposition is a scoping mechanism which allows coordination of a transaction to be delegated across a hierarchy of coordinators. See Figure 17.4, "Interpositions" for a graphical representation of this concept.
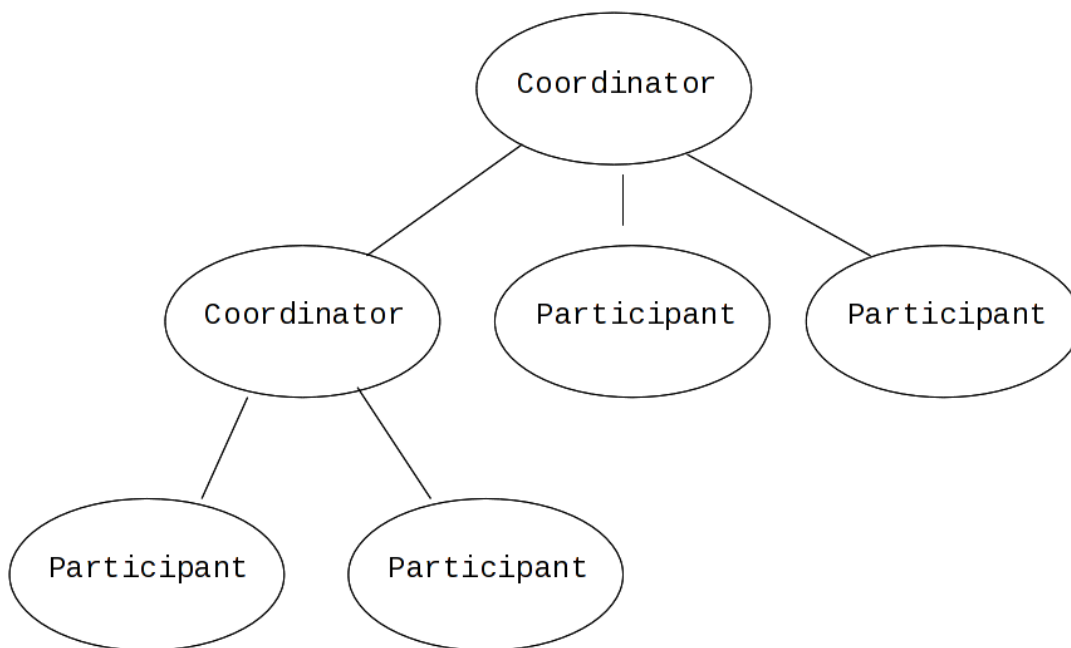
Figure 17.4. Interpositions

Interposition is particularly useful for Web Services transactions, as a way of limiting the amount of

network traffic required for coordination. For example, if communications between the top-level coordinator and a web service are slow because of network traffic or distance, the web service might benefit from executing in a subordinate transaction which employs a local coordinator service. In Figure 17.4, "Interpositions", to `prepare`, the top-level coordinator only needs to send one `prepare` message to the subordinate coordinator, and receive one `prepared` or `aborted` reply. The subordinate coordinator forwards a `prepare` locally to each participant and combines the results to decide whether to send a single `prepared` or `aborted` reply.

17.10. A NEW TRANSACTION PROTOCOL

Many component technologies offer mechanisms for coordinating ACID transactions based on two-phase commit semantics. Some of these are CORBA/OTS, JTS/JTA, and MTS/MSDTC. ACID transactions are not suitable for all Web Services transactions, as explained in Reasons ACID is Not Suitable for Web Services.

Reasons ACID is Not Suitable for Web Services

- Classic ACID transactions assume that an organization that develops and deploys applications owns the entire infrastructure for the applications. This infrastructure has traditionally taken the form of an Intranet. Ownership implies that transactions operate in a trusted and predictable manner. To assure ACIDity, potentially long-lived locks can be kept on underlying data structures during two-phase commit. Resources can be used for any period of time and released when the transaction is complete.

  In Web Services, these assumptions are no longer valid. One obvious reason is that the owners of data exposed through a Web service refuse to allow their data to be locked for extended periods, since allowing such locks invites denial-of-service attacks.

- All application infrastructures are generally owned by a single party. Systems using classical ACID transactions normally assume that participants in a transaction will obey the directives of the transaction manager and only infrequently make unilateral decisions which harm other participants in a transaction.

  Web Services participating in a transaction can effectively decide to resign from the transaction at any time, and the consumer of the service generally has little in the way of quality of service guarantees to prevent this.

17.10.1. Transaction in Loosely Coupled Systems

Extended transaction models which relax the ACID properties have been proposed over the years. WS-T provides a new transaction protocol to implement these concepts for the Web Services architecture. XTS is designed to accommodate four underlying requirements inherent in any loosely coupled architecture like Web Services. These requirements are discussed in Requirements of Web Services .

Requirements of Web Services

- Ability to handle multiple successful outcomes to a transaction, and to involve operations whose effects may not be isolated or durable.

- Coordination of autonomous parties whose relationships are governed by contracts, rather than the dictates of a central design authority.

- Discontinuous service, where parties are expected to suffer outages during their lifetimes, and coordinated work must be able to survive such outages.

- Interoperation using XML over multiple communication protocols. XTS uses SOAP encoding carried over HTTP.

CHAPTER 18. OVERVIEW OF PROTOCOLS USED BY XTS

This section discusses fundamental concepts associated with the WS-Coordination, WS-Atomic Transaction and WS-Business Activity protocols, as defined in each protocol's specification. Foundational information about these protocols is important to understanding the remaining material covered in this guide.

> **NOTE**
>
> If you are familiar with the WS-Coordination, WS-Atomic Transaction, and WS-Business Activity specifications you may only need to skim this chapter.

18.1. WS-COORDINATION

In general terms, *coordination* is the act of one entity,known as the coordinator, disseminating information to a number of participants for some domain-specific reason. This reason could be to reach consensus on a decision by a distributed transaction protocol, or to guarantee that all participants obtain a specific message, such as in a reliable multicast environment. When parties are being coordinated, information, known as the *coordination context,* is propagated to tie together operations which are logically part of the same coordinated work or activity. This context information may flow with normal application messages, or may be an explicit part of a message exchange. It is specific to the type of coordination being performed.

The fundamental idea underpinning *WS-Coordination (WS-C)* is that a coordination infrastructure is needed in a Web Services environment. The WS-C specification defines a framework that allows different coordination protocols to be plugged in to coordinate work between clients, services, and participants, as shown in Figure 18.1, "WS-C Architecture".
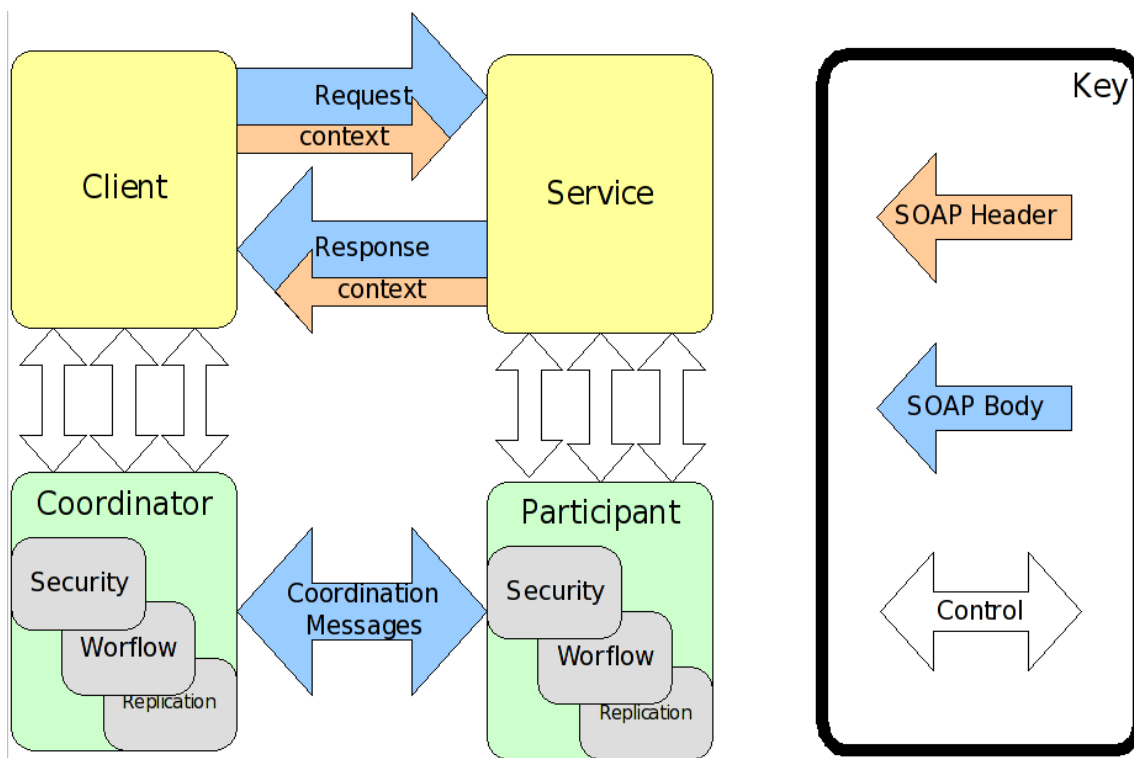

Figure 18.1. WS-C Architecture

Whatever coordination protocol is used, and in whatever domain it is deployed, the same generic requirements are present.

Generic Requirements for WS-C

- Instantiation, or activation, of a new coordinator for the specific coordination protocol, for a particular application instance.

- Registration of participants with the coordinator, such that they will receive that coordinator's protocol messages during (some part of) the application's lifetime.

- Propagation of contextual information between Web Services that comprise the application.

- An entity to drive the coordination protocol through to completion.

The first three of the points in Generic Requirements for WS-C are the direct responsibility of WS-C, while the fourth is the responsibility of a third-party entity. The third-party entity is usually the client component of the overall application. These four WS-C roles and their relationships are shown in Figure 18.2, "Four Roles in WS-C".
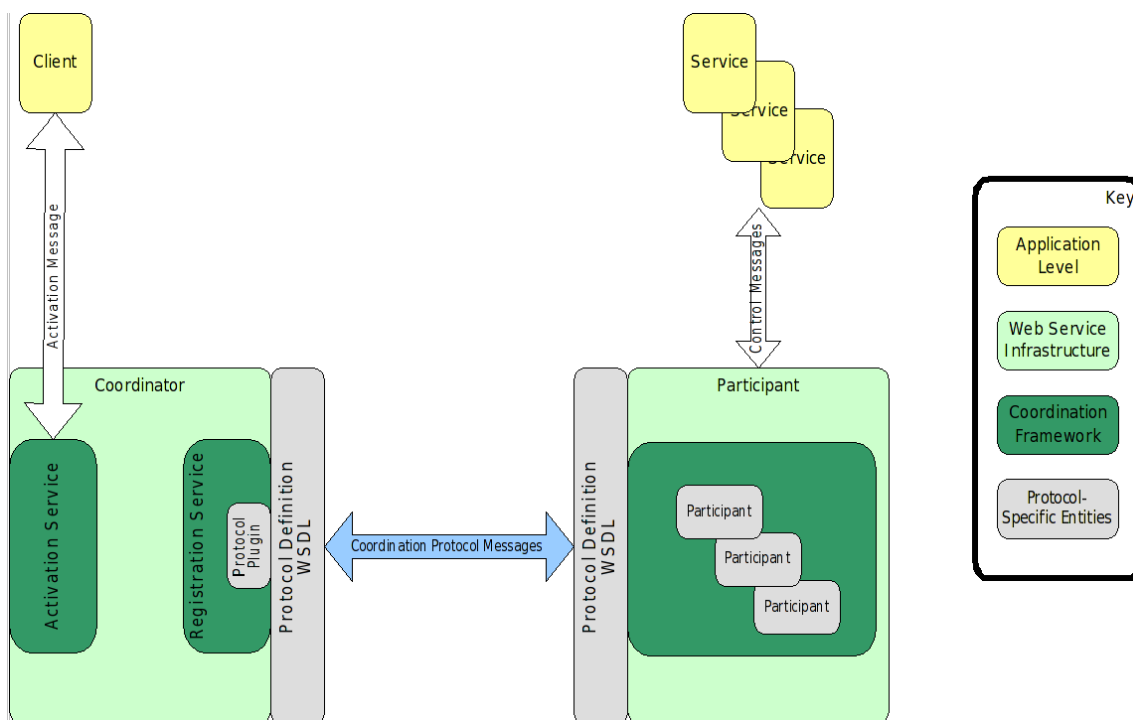
Figure 18.2. Four Roles in WS-C

18.1.1. Activation

The WS-C framework exposes an Activation Service which supports the creation of coordinators for specific coordination protocols and retrieval of associated contexts. Activation services are invoked synchronously using an RPC style exchange. So, the service WSDL defines a single port declaring a `CreateCoordinationContext` operation. This operation takes an input specfying the details of the transaction to be created, including the type of coordination required, timeout, and other relevant information. It returns an output containing the details of the newly-created transaction context: the transaction identifier, coordination type, and registration service URL.

Example 18.1.

```
<!-- Activation Service portType Declaration -->
<wsdl:portType name="ActivationCoordinatorPortType">
  <wsdl:operation name="CreateCoordinationContext">
    <wsdl:input message="wscoor:CreateCoordinationContext"/>
```

```
        <wsdl:output message="wscoor:CreateCoordinationContextResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

NOTE

The 1.0 Activation Coordinator service employs an asynchronous message exchange comprised of two one-way messages, so an Activation Requester service is also necessary.

18.1.2. Registration

The context returned by the activation service includes the URL of a Registration Service. When a web service receieves a service request accompanied by a transaction context, it contacts the Registration Service to enroll as a participant in the transaction. The registration request includes a participant protocol defining the role the web service wishes to take in the transaction. Depending upon the coordination protocol, more than one choice of participant protocol may be available.

Like the activation service, the registration service assumes synchronous communication. Thus, the service WSDL exposes a single port declaring a `Register` operation. This operation takes an input specifying the details of the participant which is to be registered, including the participant protocol type. It returns a corresponding output response.

Example 18.2. Registration ServiceWSDL Interface

```
<!-- Registration Service portType Declaration -->
<wsdl:portType name="RegistrationCoordinatorPortType">
  <wsdl:operation name="Register">
    <wsdl:input message="wscoor:Register"/>
    <wsdl:output message="wscoor:RegisterResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

Once a participant is registered with a coordinator through the registration service, it receives coordination messages from the coordinator. Typical messages include such things as "prepare to complete" and "complete" messages, if a two-phase protocol is used. Where the coordinator's protocol supports it, participants can also send messages back to the coordinator.

NOTE

The 1.0 Registration Coordinator service employs an asynchronous message exchange comprised of two one way messages, so a Registration Requester service is also necessary

18.1.3. Completion

The role of terminator is generally filled by the client application. At an appropriate point, the client asks the coordinator to perform its particular coordination function with any registered participants, to drive the protocol through to its completion. After completion, the client application may be informed of an outcome for the activity. This outcome may take any form along the spectrum from simple success or failure notification, to complex structured data detailing the activity's status.
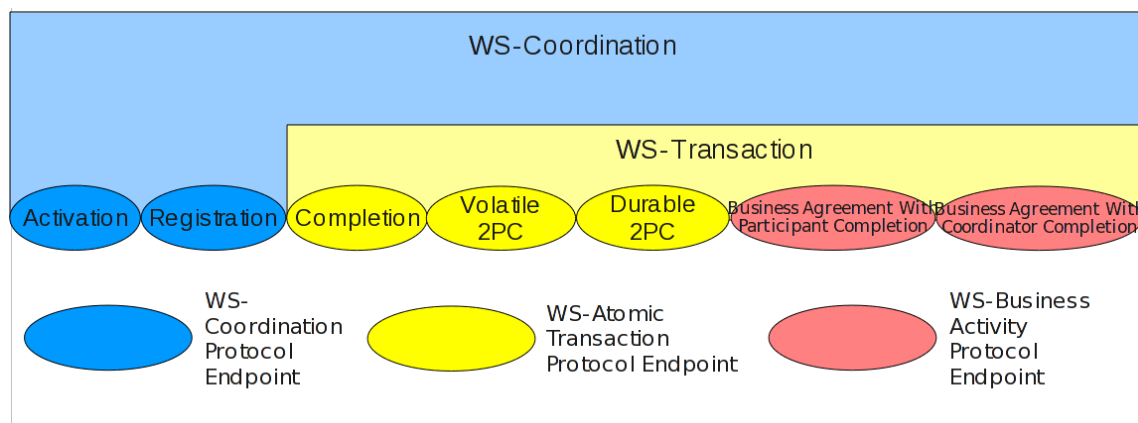
18.2. WS-TRANSACTION

WS-Transaction (WS-T) comprises the pair of transaction coordination protocols, *WS-Atomic Transaction (WS-AT)* and *WS-Business Activity (WS-BA)*, which utilize the coordination framework provided by *WS-Coordination (WS-C)*.

WS-Transactions was developed to unify existing traditional transaction processing systems, allowing them to communicate reliably with one another without changes to the systems' own function.

18.2.1. WS-Transaction Foundations

WS-Transaction is layered upon the WS-Coordination protocol, as shown in as shown in Figure 18.3, "WS-Coordination, WS-Transaction, and WS-Business Activity".

Figure 18.3. WS-Coordination, WS-Transaction, and WS-Business Activity

WS-C provides a generic framework for specific coordination protocols, like WS-Transaction, used in a modular fashion. WS-C provides only context management, allowing contexts to be created and activities to be registered with those contexts. WS-Transaction leverages the context management framework provided by WS-C in two ways.

1. It extends the WS-C context to create a transaction context.

2. It augments the activation and registration services with a number of additional services (Completion, Volatile2PC, Durable2PC, BusinessAgreementWithParticipantCompletion, and BusinessAgreementWithCoordinatorCompletion) and two protocol message sets (one for each of the transaction models supported in WS-Transaction), to build a fully-fledged transaction coordinator on top of the WS-C protocol infrastructure.

3. An important aspect of WS-Transaction that differs from traditional transaction protocols is that a synchronous request/response model is not assumed. Sequences of one way messages are used to implement communications between the client/participant and the coordination services appropriate to the transaction's coordination and participant protocols. This is significant because it means that the client and participant containers must deploy XTS service endpoints to receive messages from the coordinator service.

   This requirement is visible in the details of the **Register** and **RegisterResponse** messages declared in the Registration Service WSDL in Example 18.2, "Registration ServiceWSDL Interface". The **Register** message contains the URL of an endpoint in the client or web service container. This URL is used when a WS-Transaction coordination service wishes to dispatch a message to the client or web service. Similarly, the **RegisterResponse** message

contains a URL identifying an endpoint for the protocol-specific WS-Transaction coordination service for which the client/web service is registered, allowing messages to be addressed to the transaction coordinator.

18.2.2. WS-Transaction Architecture

WS-Transaction distnguishes the transaction-aware web service in its role executing business-logic, from the web service acting as a participant in the transaction, communicating with and responding to its transaction coordinator. Transaction-aware web services deal with application clients using business-level protocols, while the participant handles the underlying WS-Transaction protocols, as shown in Figure 18.4, "WS-Transaction Global View" .
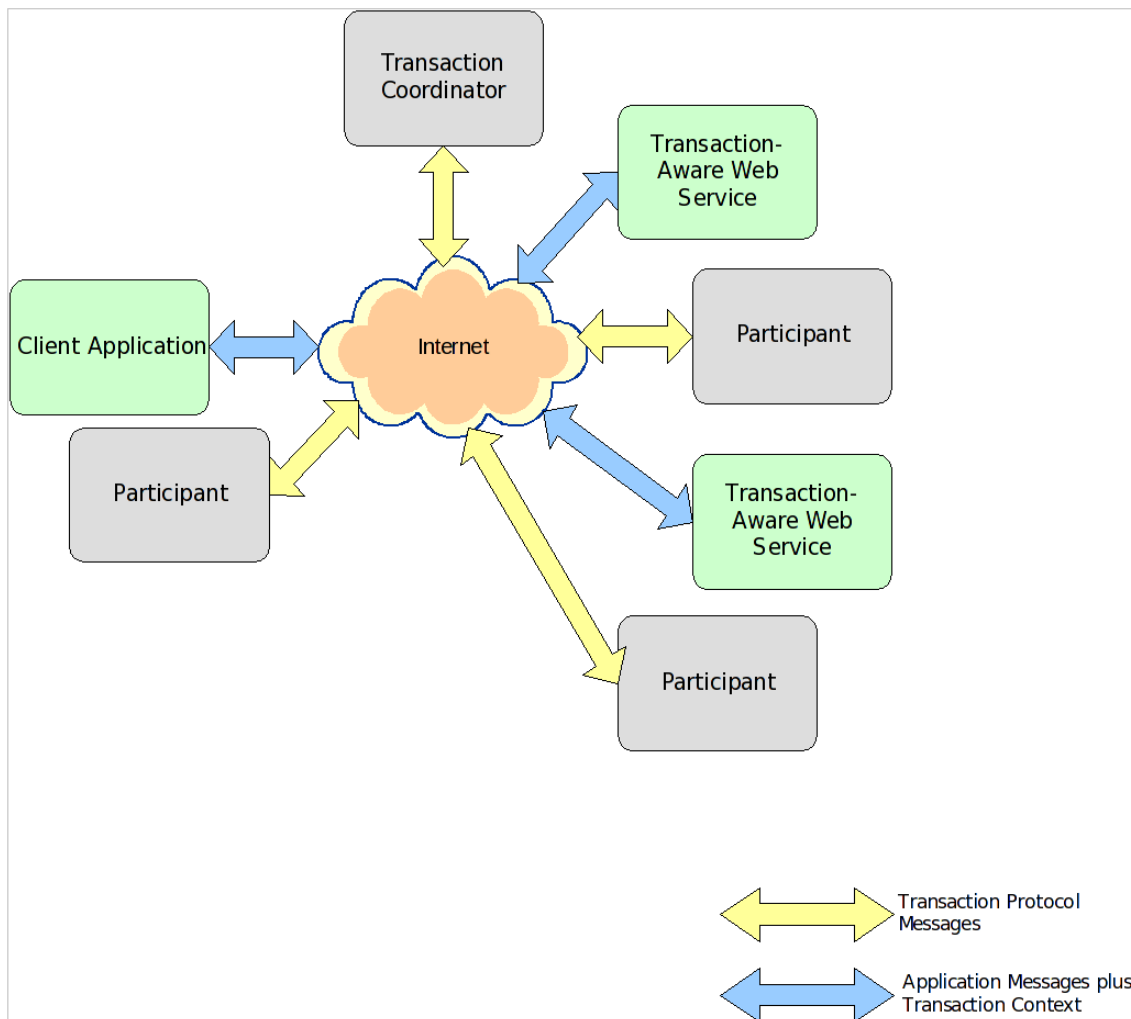
Figure 18.4. WS-Transaction Global View

A transaction-aware web service encapsulates the business logic or work that needs to be conducted within the scope of a transaction. This work cannot be confirmed by the application unless the transaction also commits. Thus, control is ultimately removed from the application and given to the transaction.

The participant is the entity that, under the dictates of the transaction coordinator, controls the outcome of the work performed by the transaction-aware Web service. In Figure 18.4, "WS-Transaction Global View", each web service is shown with one associated participant that manages the transaction protocol messages on behalf of its web service. Figure 18.5, "WS-Transaction Web Services and Participants", however, shows a close-up view of a single web service, and a client application with their associated participants.
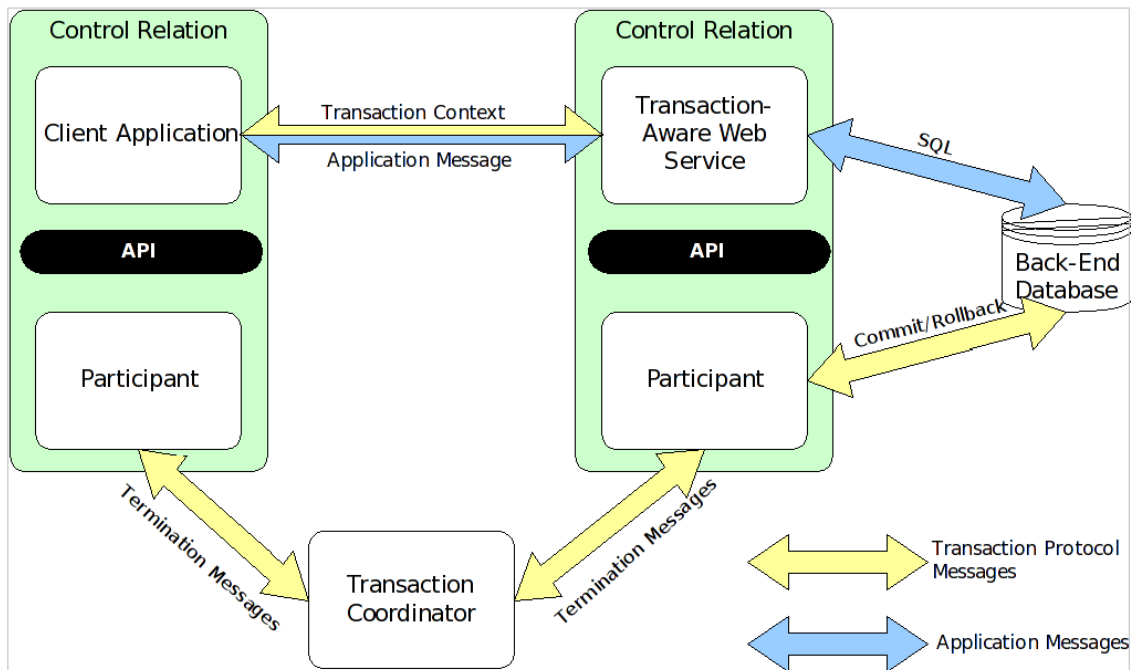
Figure 18.5. WS-Transaction Web Services and Participants

The transaction-aware web service employs a back end database accessed via a JDBC driver, which sends SQL statements to the database for processing. However, those statements should only commit if the enclosing web service transaction does. For this to work, the web service must employ transaction bridging. Transaction bridging registers a participant with the coordinator for the web service transaction and creates a matching XA transaction within which it can invoke the driver to make tentative changes to the database. The web service ensures that service requests associated with a specific web service transaction are executed in the scope of the corresponding XA transaction, grouping changes common to a given transaction while isolating changes belonging to different transactions. The participant responds to prepare, commit, or rollback requests associated from the web service transaction coordinator by forwarding the same operations to the underlying XA transaction coordinator, ensuring that the local outcome in the database corresponds with the global outcome of the web service transaction as a whole.

Things are less complex for the client. Through its API, the client application registers a participant with the transaction, and uses this participant to control termination of the transaction.

18.2.3. WS_Transaction Models

It has been established that traditional transaction models are not appropriate for Web Services. No one specific protocol is likely to be sufficient, given the wide range of situations where Web service transactions are likely to be used. The WS-Transaction specification proposes two distinct models, where each supports the semantics of a particular kind of B2B interaction.

The following discussion presents the interactions between the client, web service and the transaction coordinator in great detail for expository purposes only. Most of this activity happens automatically behind the scenes. The actual APIs used to initiate and complete a transaction and to register a participant and drive it through the commit or abort process are described in Chapter 21, *The XTS API*.

18.2.3.1. Atomic Transactions

An *atomic transaction (AT)* is similar to traditional ACID transactions, and is designed to support short-duration interactions where ACID semantics are appropriate. Within the scope of an AT, web services typically employ bridging to allow them to access XA resources, such as databases and message

queues, under the control of the web service transaction. When the transaction terminates, the participant propagates the outcome decision of the AT to the XA resources, and the appropriate commit or rollback actions are taken by each.

All services and associated participants are expected to provide ACID semantics, and it is expected that any use of atomic transactions occurs in environments and situations where ACID is appropriate. Usually, this environment is a trusted domain, over short durations.

Procedure 18.1. Atomic Transaction Process

1. To begin an atomic transaction, the client application first locates a WS-C Activation Coordinator web service that supports WS-Transaction.

2. The client sends a WS-C `CreateCoordinationContext` message to the service, specifying http://schemas.xmlsoap.org/ws/2004/10/wsat as its coordination type.

3. The client receives an appropriate WS-Transaction context from the activation service.

4. The response to the `CreateCoordinationContext` message, the transaction context, has its `CoordinationType` element set to the WS-Atomic Transaction namespace, http://schemas.xmlsoap.org/ws/2004/10/wsat. It also contains a reference to the atomic transaction coordinator endpoint, the WS-C Registration Service, where participants can be enlisted.

5. The client normally proceeds to invoke Web Services and complete the transaction, either committing all the changes made by the web services, or rolling them back. In order to be able to drive this completion activity, the client must register itself as a participant for the `Completion` protocol, by sending a `Register` message to the Registration Service whose endpoint was returned in the Coordination Context.

6. Once registered for Completion, the client application then interacts with Web Services to accomplish its business-level work. With each invocation of a business Web service, the client inserts the transaction context into a SOAP header block, such that each invocation is implicitly scoped by the transaction. The toolkits that support WS-Atomic Transaction-aware Web Services provide facilities to correlate contexts found in SOAP header blocks with back-end operations. This ensures that modifications made by the Web service are done within the scope of the same transaction as the client and subject to commit or rollback by the transaction coordinator.

7. Once all the necessary application-level work is complete, the client can terminate the transaction, with the intent of making any changes to the service state permanent. The completion participant instructs the coordinator to try to commit or roll back the transaction. When the commit or roll-back operation completes, a status is returned to the participant to indicate the outcome of the transaction.

Although this description of the completion protocol seems straightforward, it hides the fact that in order to resolve the transaction to an outcome, several other participant protocols need to be followed.

Volatile2pc

The first of these protocols is the optional *Volatile2PC* (2PC is an abbreviation referring to the two-phase commit). The Volatile2PC protocol is the WS-Atomic Transaction equivalent of the synchronization protocol discussed earlier. It is typically executed where a Web service needs to flush volatile (cached) state, which may be used to improve performance of an application, to a database prior to the transaction committing. Once flushed, the data is controlled by a two-phase aware participant.

When the completion participant initiates a `commit` operation, all Volatile2PC participants are informed that the transaction is about to complete, via the `prepare` message. The participants can respond with one of three messages: `prepared`, `aborted`, or `readonly`. A failure at this stage causes the transaction to roll back.

Durable2PC

The next protocol in the WS-Atomic Transaction is *Durable2PC*. The Durable2PC protocol is at the core of WS-Atomic Transaction. It brings about the necessary consensus between participants in a transaction, so the transaction can safely be terminated.

The Durable2PC protocol ensures atomicity between participants, and is based on the classic technique of two-phase commit with presumed abort .

Procedure 18.2. Durable2PC Procedure

1. During the first phase, when the coordinator sends the prepare message, a participant must make durable any state changes that occurred during the scope of the transaction, so these changes can either be rolled back or committed later. None of the original state information can be lost at this point, since the atomic transaction may still roll back. If the participant cannot `prepare`, it must inform the coordinator, by means of the `aborted` message. The transaction will ultimately roll back. If the participant is responsible for a service that did not change any of the transaction's data,, it can return the `readonly` message, causing it to be omitted from the second phase of the commit protocol. Otherwise, the `prepared` message is sent by the participant.

2. If no failures occur during the first phase, Durable2PC proceeds to the second phase, in which the coordinator sends the `commit` message to participants. Participants then make permanent the tentative work done by their associated services, and send a `committed` message to the coordinator. If any failures occur, the coordinator sends the `rollback` message to all participants, causing them to discard tentative work done by their associated services, and delete any state information saved to persistent storage at `prepare`, if they have reached that stage. Participants respond to a rollback by sending an `aborted` message to the coordinator.
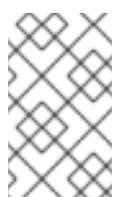
> **NOTE**
>
> The semantics of the WS-Atomic Transaction protocol do not include the one-phase commit optimization. A full two-phase commit is always used, even where only a single participant is enlisted.

Figure 18.6, "WS-Atomic Two-Phase Participant State Transitions" shows the state transitions of a WS-Atomic Transaction and the message exchanges between coordinator and participant. Messages generated by the coordinator are represented by solid lines, while the participants' messages use dashed lines.
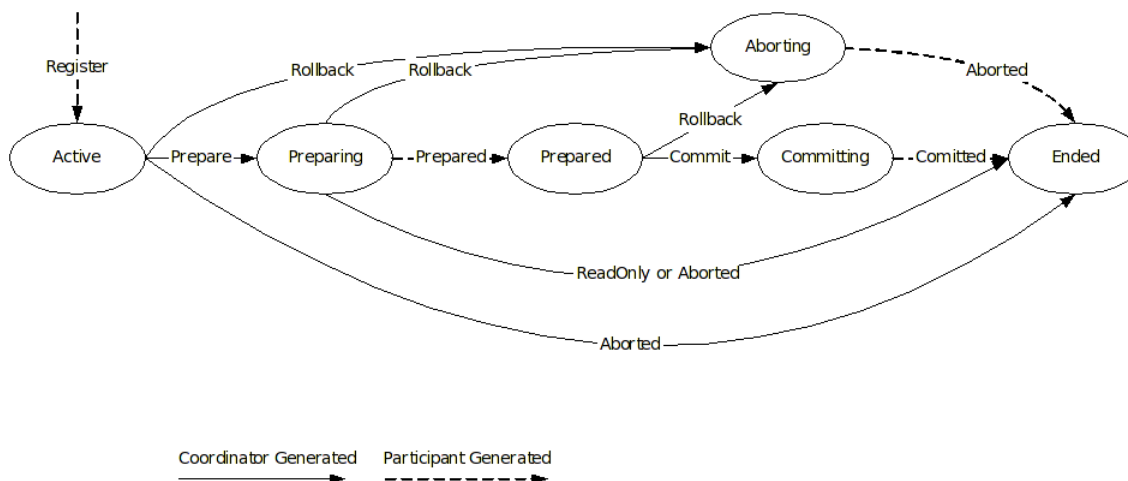
Figure 18.6. WS-Atomic Two-Phase Participant State Transitions

Once the Durable2PC protocol completes, the `Completion` protocol that originally began the termination of the transaction can complete, and inform the client application whether the transaction was committed or rolled back. Additionally, the Volatile2PC protocol may complete.

Like the `prepare` phase of Volatile2PC, the final phase is optional and can be used to inform participants about the transaction's completion, so that they can release resources such as database connections.

Any registered Volatile2PC participants are invoked after the transaction terminates, and are informed about the transaction's completion state by the coordinator. Since the transaction has terminated, any failures of participants at this stage are ignored, since they have no impact on outcomes.

Figure 18.7, "" illustrates the intricate interweaving of individual protocols comprising the AT as a whole.
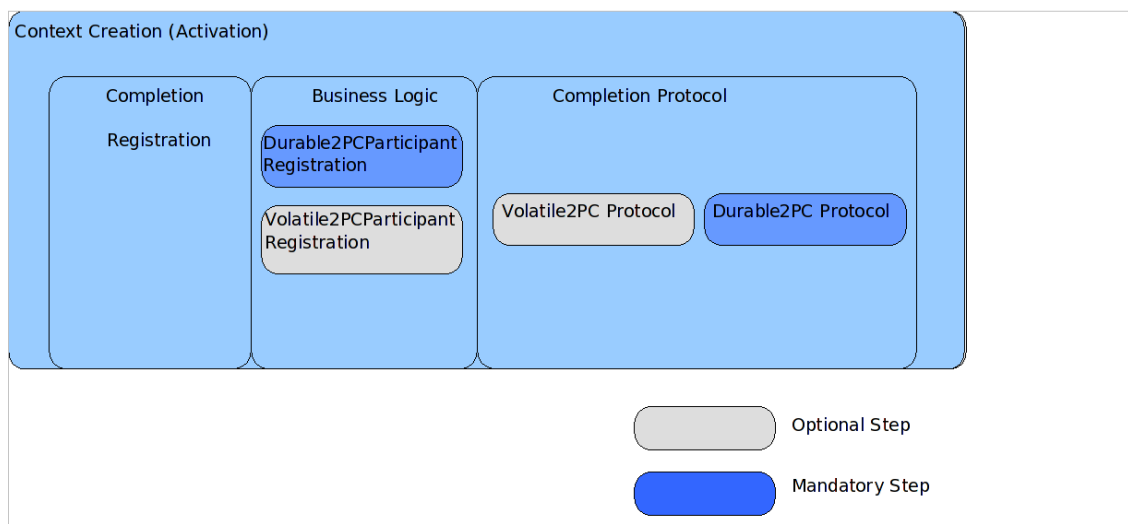
Figure 18.7.

18.2.3.2. Business Activities

Most B2B applications require transactional support in order to guarantee consistent outcome and correct execution. These applications often involve long-running computations, loosely coupled systems, and components that do not share data, location, or administration. It is difficult to incorporate atomic transactions within such architectures.

For example, an online bookshop may reserve books for an individual for a specific period of time. However, if the individual does not purchase the books within that period, they become available again for purchase by other customers. Because it is not possible to have an infinite supply of stock, some online shops may seem, from the user's perspective, to reserve items for them, while actually allow others to preempt the reservation. A user may discover, to his disappointment, that the item is no longer available.

A *Business Activity (BA)*is designed specifically for these kinds of long-duration interactions, where it is impossible or impractical to exclusively lock resources.

Procedure 18.3. BA Process Overview

1. Services are requested to do work.

2. Where those services have the ability to undo any work, they inform the BA, in case the BA later decides the cancel the work. If the BA suffers a failure. it can instruct the service to execute its **undo** behavior.

The key to BA is that how services do their work and provide compensation mechanisms is not the responsibility of the WS-BA specification. It is delegated to the service provider.

The WS-BA defines a protocol for Web Services-based applications to enable existing business processing and work-flow systems to wrap their proprietary mechanisms and interoperate across implementations and business boundaries.

Unlike the WS-AT protocol model, where participants inform the coordinator of their state only when asked, a child activity within a BA can specify its outcome to the coordinator directly, without waiting for a request. A participant may choose to exit the activity or may notify the coordinator of a failure at any point. This feature is useful when tasks fail, since the notification can be used to modify the goals and drive processing forward, without the need to wait until the end of the transaction to identify failures. A well-designed Business Activity should be proactive.

The BA protocols employ a compensation-based transaction model. When a participant in a business activity completes its work, it may choose to exit the activity. This choice does not allow any subsequent rollback. Alternatively, the participant can complete its activity, signaling to the coordinator that the work it has done can be compensated if, at some later point, another participant notifies a failure to the coordinator. In this latter case, the coordinator asks each non-exited participant to compensate for the failure, giving them the opportunity to execute whatever compensating action they consider appropriate. For instance, participant might credit a bank account which it previously debited. If all participants exit or complete without failure, the coordinator notifies each completed participant that the activity has been closed.

Underpinning all of this are three fundamental assumptions, detailed in Assumptions of WS-BA.

Assumptions of WS-BA

- All state transitions are reliably recorded, including application state and coordination metadata (the record of sent and received messages).

- All request messages are acknowledged, so that problems are detected as early as possible. This avoids executing unnecessary tasks and can also detect a problem earlier when rectifying it is simpler and less expensive.

- As with atomic transactions, a *response* is defined as a separate operation, not as the output of the request. Message I/O implementations typically have timeout requirements too short for BA responses. If the response is not received after a timeout, it is re-sent, repeatedly, until a

response is received. The receiver discards all but one identical request received.

The BA model has two participant protocols: **BusinessAgreementWithParticipantCompletion** and **BusinessAgreementWithCoordinatorCompletion**. Unlike the AT protocols which are driven from the coordinator down to participants, this protocol takes the opposite approach.

BusinessAgreementWithParticipantCompletion

1. A participant is initially created in the Active state.

2. If it finishes its work and it is no longer needed within the scope of the BA (such as when the activity operates on immutable data), the participant can unilaterally decide to exit, sending an **exited** message to the coordinator. However, if the participant finishes and wishes to continue in the BA, it must be able to compensate for the work it has performed. In this case, it sends a **completed** message to the coordinator and waits for the coordinator to notify it about the final outcome of the BA. This outcome is either a **close** message, meaning the BA has completed successfully, or a **compensate** message indicating that the participant needs to reverse its work.

BusinessAgreementWithCoordinatorCompletion

The **BusinessAgreementWithCoordinatorCompletion** differs from the **BusinessAgreementWithParticipantCompletion** protocol in that the participant cannot autonomously decide to complete its participation in the BA, even if it can be compensated.

1. Instead, the completion stage is driven by the client which created the BA, which sends a **completed** message to the coordinator.

2. The coordinator sends a **complete** message to each participant, indicating that no further requests will be sent to the service associated with the participant.

3. The participant continues on in the same manner as in the BusinessAgreementWithParticipantCompletion protocol.

The advantage of the BA model, compared to the AT model, is that it allows the participation of services that cannot lock resources for extended periods.

While the full ACID semantics are not maintained by a BA, consistency can still be maintained through compensation. The task of writing correct compensating actions to preserve overall system consistency is the responsibility of the developers of the individual services under control of the BA. Such compensations may use backward error recovery, but forward recovery is more common.

Figure 18.8, "" shows the state transitions of a WS-BA **BusinessAgreementWithParticipantCompletion** participant and the message exchanges between coordinator and participant. Messages generated by the coordinator are shown with solid lines, while the participants' messages are illustrated with dashed lines.
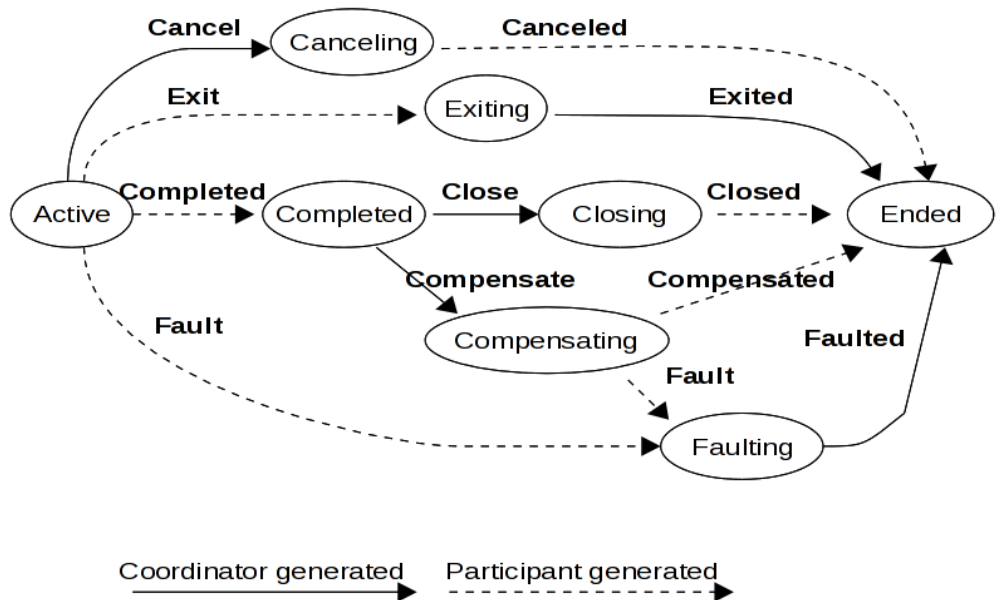
Figure 18.8.

Figure 18.9, "" shows the state transitions of a WS-BA
BusinessAgreementWithCoordinatorCompletion participant and the message exchanges
between coordinator and participant. Messages generated by the coordinator are shown with solid
lines, while the participants' messages are illustrated with dashed lines.
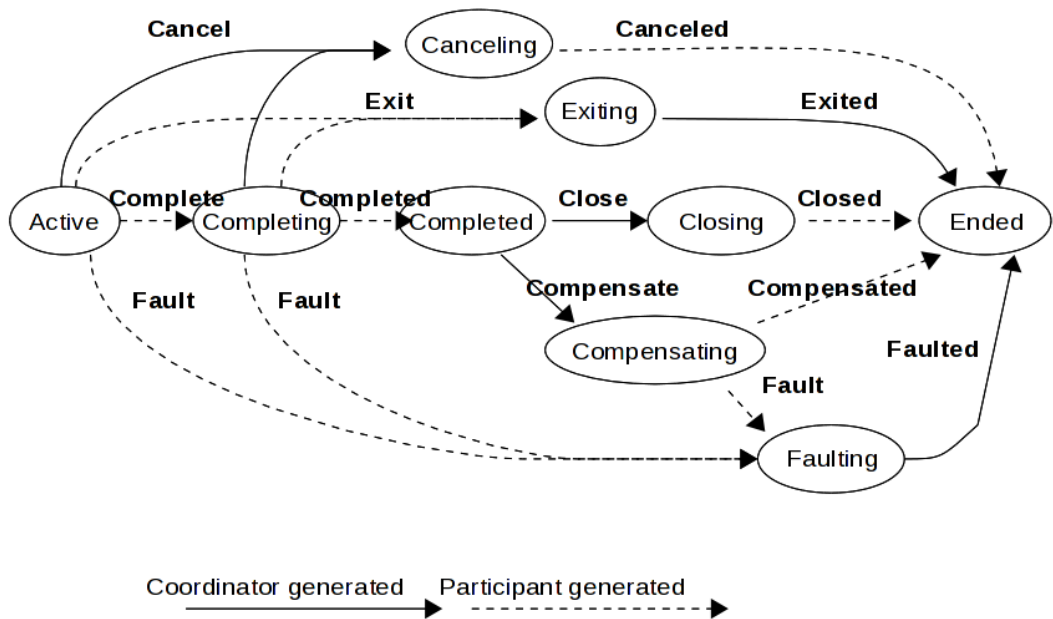
Figure 18.9.

18.2.4. Application Messages

Application messages are the requests and responses sent between parties, that constitute the work of
a business process. Any such messages are considered opaque by XTS, and there is no mandatory
message format, protocol binding, or encoding style. This means that you are free to use any
appropriate Web Services protocol. In XTS, the transaction context is propagated within the headers of
SOAP messages.

XTS ships with support for service developers building WS-Transactions-aware services on the JBoss

Enterprise Application Platform. Interceptors are provided for automatic context handling at both client and service, which significantly simplifies development, allowing you to concentrate on writing the business logic without being sidetracked by the transactional infrastructure. The interceptors add and remove context elements to application messages, without altering the semantics of the messages themselves. Any service which understands what to do with a WS-C context can use it. Services which are not aware of WS-C, WS-Atomic Transaction and WS-Business Activity can ignore the context. XTS manages contexts without user intervention.

18.2.4.1. WS-C, WS-Atomic Transaction, and WS-Business Activity Messages

Although the application or service developer is rarely interested in the messages exchanged by the transactional infrastructure, it is useful to understand what kinds of exchanges occur so that the underlying model can be fitted in to an overall architecture.

WS-Coordination, WS-Atomic Transaction and WS-Business Activity-specific messages are transported using SOAP messaging over HTTP. The types of messages that are propagated include instructions to perform standard transaction operations like `begin` and `prepare`.

> **NOTE**
>
> XTS messages do not interfere with messages from the application, an application need not use the same transport as the transaction-specific messages. For example, a client application might deliver its application-specific messages using SOAP RPC over SMTP, even though the XTS messages are delivered using a different mechanism.

18.3. SUMMARY

XTS provides a coordination infrastructure which allows transactions to run between services owned by different business, across the Internet. That infrastructure is based on the WS-C, WS-Atomic Transaction and WS-Business Activity specifications. It supports two kinds of transactions: atomic transactions and business activities, which can be combined in arbitrary ways to map elegantly onto the transactional requirements of the underlying problem. The use of the whole infrastructure is simple, because its functionality is exposed through a simple transactioning API. XTS provides everything necessary to keep application and transactional aspects of an application separate, and to ensure that a system's use of transactions does not interfere with the functional aspects of the system itself.

CHAPTER 19. GETTING STARTED

19.1. INSTALLING THE XTS SERVICE ARCHIVE INTO JBOSS TRANSACTION SERVICE

XTS, which is the Web Services component of JBoss Transaction Service, provides WS-AT and WS-BA support for Web Services hosted on the Enterprise Application Platform. The module is packaged as a *Service Archive (.sar)* located in **$JBOSS_HOME/docs/examples/transactions/**. To install it, follow Procedure 19.1, "Installing the XTS Module".

Procedure 19.1. Installing the XTS Module

1. Create a sub-directory in the **$JBOSS_HOME/server/[*name*]/deploy/** directory, called **jbossxts.sar/**.

2. Unpack the SAR, which is a ZIP archive, into this new directory.

3. Restart JBoss Enterprise Application Platform to activate the module.

19.2. CREATING CLIENT APPLICATIONS

There are two aspects to a client application using XTS, the transaction declaration aspects, and the business logic. The business logic includes the invocation of Web Services.

Transaction declaration aspects are handled automatically with the XTS client API. This API provides simple transaction directives such as **begin**, **commit**, and **rollback**, which the client application can use to initialize, manage, and terminate transactions. Internally, this API uses SOAP to invoke operations on the various WS-C, WS-AT and WS-BA services, in order to create a coordinator and drive the transaction to completion.

19.2.1. User Transactions

A client uses the **UserTransactionFactory** and **UserTransaction** classes to create and manage WS-AT transactions. These classes provide a simple API which operates in a manner similar to the JTA API. A WS-AT transaction is started and associated with the client thread by calling the **begin** method of the **UserTransaction** class. The transaction can be committed by calling the **commit** method, and rolled back by calling the **rollback** method.

More complex transaction management, such as suspension and resumption of transactions, is supported by the **TransactionManagerFactory** and **TransactionManager** classes.

Full details of the WS-AT APIs are provided in Chapter 21, *The XTS API*.

19.2.2. Business Activities

A client creates and manages Business Activities using the **UserBusinessActivityFactory** and **UserBusinessActivity** classes. A WS-BA activity is started and associated with the client thread by calling the **begin** method of the **UserBusinessActivity** class. A client can terminate a business activity by calling the **close** method, and cancel it by calling the **cancel** method.

If any of the Web Services invoked by the client register for the **BusinessActivityWithCoordinatorCompletion** protocol, the client can call the **completed** method before calling the **close** method, to notify the services that it has finished making service

invocations in the current activity.

More complex business activity management, such as suspension and resumption of business activities, is supported by the `BusinessActivityManagerFactory` and `BusinessActivityManager` classes.

Full details of the WS-AT APIs are provided in Chapter 21, *The XTS API*.

19.2.3. Client-Side Handler Configuration

XTS does not require the client application to use a specific API to perform invocations on transactional Web Services. The client is free to use any appropriate API to send SOAP messages to the server and receive SOAP responses. The only requirements imposed on the client are:

- It must forward details of the current transaction to the server when invoking a web service.

- It must process any responses from the server in the context of the correct transaction.

In order to achieve this, the client must insert details of the current XTS context into the headers of outgoing SOAP messages, and extract the context details from the headers of incoming messages and associate the context with the current thread. To simplify this process, the XTS module includes handlers which can perform this task automatically. These handlers are designed to work with JAX-WS clients.

> **NOTE**
>
> If you choose to use a different SOAP client/server infrastructure for business service invocations, you must provide for header processing. XTS only provides interceptors for or JAX-WS. A JAX-RPC handler is provided only for the 1.0 implementation.

19.2.3.1. JAX-WS Client Context Handlers

In order to register the JAX-WS client-side context handler, the client application uses the APIs provided by the `javax.xml.ws.BindingProvider` and `javax.xml.ws.Binding` classes, to install a handler chain on the service proxy which is used to invoke the remote endpoint. Refer to the example application client implementation located in the `src/com/jboss/jbosstm/xts/demo/BasicClient.java` file for an example.

You can also specify the handlers by using a configuration file deployed with the application. The file is identified by attaching a `javax.jws.HandlerChain` annotation to the interface class, which declares the JAX-WS client API. This interface is normally generated from the web service WSDL port definition.

You need to instantiate the `com.arjuna.mw.wst11.client.JaxWSHeaderContextProcessor` class when registering a JAX-WS client context handler.

19.3. CREATING TRANSACTIONAL WEB SERVICES

The two parts to implementing a Web service using XTS are the transaction management and the business logic.

The bulk of the transaction management aspects are organized in a clear and easy-to-implement model by means of the XTS's *Participant API*, provides a structured model for negotiation between the web service and the transaction coordinator. It allows the web service to manage its own local transactional data, in accordance with the needs of the business logic, while ensuring that its activities

are in step with those of the client and other services involved in the transaction. Internally, this API uses SOAP to invokes operations on the various WS-C, WS-AT and WS-BA services, to drive the transaction to completion.

19.3.1. Participants

A *participant* is a software entity which is driven by the transaction manager on behalf of a Web service. When a web service wants to participate in a particular transaction, it must enroll a participant to act as a proxy for the service in subsequent negotiations with the coordinator. The participant implements an API appropriate to the type of transaction it is enrolled in, and the participant model selected when it is enrolled. For example, a Durable2PC participant, as part of a WS-Atomic Transaction, implements the Durable2PCParticipant interface. The use of participants allows the transactional control management aspects of the Web service to be factored into the participant implementation, while staying separate from the the rest of the Web service's business logic and private transactional data management.

The creation of participants is not trivial, since they ultimately reflect the state of a Web service's back-end processing facilities, an aspect normally associated with an enterprise's own IT infrastructure. Implementations must use one of the following interfaces, depending upon the protocol it will participate within: `com.arjuna.wst11.Durable2PCParticipant`, `com.arjuna.wst11.Volatile2PCParticipant`, `com.arjuna.wst11.BusinessAgreementWithParticipantCompletionParticipant`, or `com.arjuna.wst11.BusinessAgreementWithCoordinatorCompletionParticipant`.

19.3.2. Service-Side Handler Configuration

A transactional Web service must ensure that a service invocation is included in the appropriate transaction. This usually only affects the operation of the participants and has no impact on the operation of the rest of the Web service. XTS simplifies this task and decouples it from the business logic, in much the same way as for transactional clients . XTS provides a handler which detects and extracts the context details from the headers in incoming SOAP headers, and associates the web service thread with the transaction. The handler clears this association when dispatching SOAP responses, and writes the context into the outgoing message headers. This is shown in Figure 19.1, "Context Handlers Registered with the SOAP Server".

The service side handlers for JAX-WS come in two different versions. The normal handler resumes any transaction identified by an incoming context when the service is invoked, and suspends this transaction when the service call completes. The alternative handler is used to interpose a local coordinator. The first time an incoming parent context is seen, the local coordinator service creates a subordinate transaction, which is resumed before the web service is called. The handler ensures that this subordinate transaction is resumed each time the service is invoked with the same parent context. When the subordinate transaction completes, the association between the parent transaction and its subordinate is cleared.

> **NOTE**
>
> The subordinate service side handler is only able to interpose a subordinate coordinator for an Atomic Transaction.

> **NOTE**
>
> JAX-RPC is provided for the 1.0 implementation only.

19.3.2.1. JAX-WS Service Context Handlers

To register the JAX-WS server-side context handler with the deployed Web Services, you must install a handler chain on the Server Endpoint Implementation class. The endpoint implementation class annotation, which is the one annotated with a `javax.jws.WebService`, must be supplemented with a `javax.jws.HandlerChain` annotation which identifies a handler configuration file deployed with the application. Please refer to the example application configuration file located at `dd/jboss/context-handlers.xml` and the endpoint implementation classes located in `src/com/jboss/jbosstm/xts/demo/services` for an example.

When registering a normal JAX-WS service context handler, you must instantiate the `com.arjuna.mw.wst11.service.JaxWSHeaderContextProcessor` class. If you need coordinator interposition, employ the `com.arjuna.mw.wst11.service.JaxWSSubordinateHeaderContextProcessor` instead.
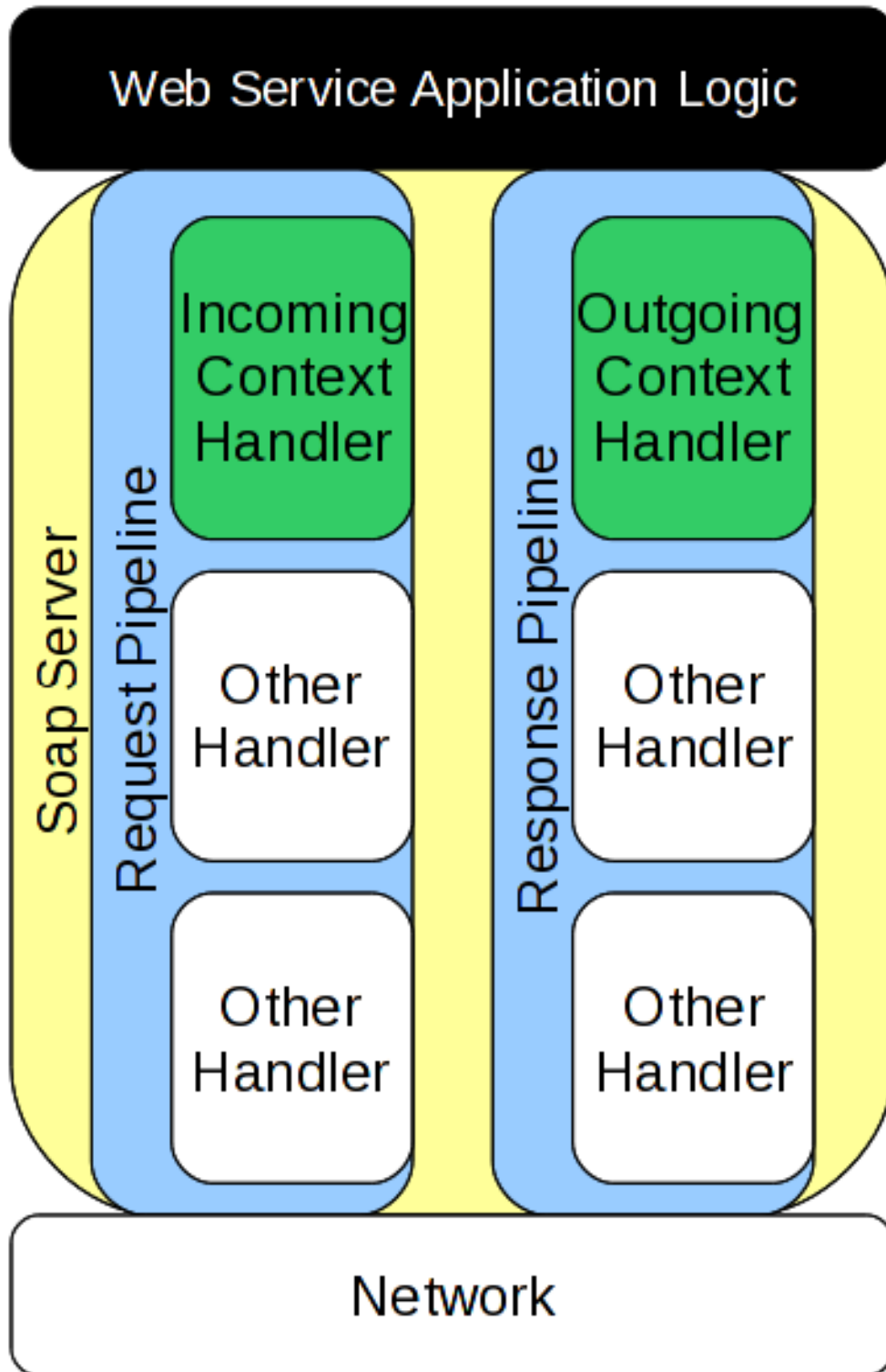
Figure 19.1. Context Handlers Registered with the SOAP Server

19.4. SUMMARY

This chapter gives a high-level overview of each of the major software pieces used by the Web Services transactions component of JBoss Transaction Service. The Web Services transaction manager provided by JBoss Transaction Service is the hub of the architecture and is the only piece of software that user-level software does not bind to directly. XTS provides header-processing infrastructure for use with Web Services transactions contexts for both client applications and Web Services. XTS provides a simple interface for developing transaction participants, along with the necessary document-handling code.

This chapter is only an overview, and does not address the more difficult and subtle aspects of programming Web Services. For fuller explanations of the components, please continue reading.

CHAPTER 20. PARTICIPANTS

20.1. OVERVIEW

The *participant* is the entity that performs the work pertaining to transaction management on behalf of the business services involved in an application. The Web service (in the example code, a theater booking system) contains some business logic to reserve a seat and inquire about availability, but it needs to be supported by something that maintains information in a durable manner. Typically this is a database, but it could be a file system, NVRAM, or other storage mechanism.

Although the service may talk to the back-end database directly, it cannot commit or undo any changes, since committing and rolling back are ultimately under the control of a transaction. For the transaction to exercise this control, it must communicate with the database. In XTS, participant does this communication, as shown in Figure 20.1, "Transactions, Participants, and Back-End Transaction Control".
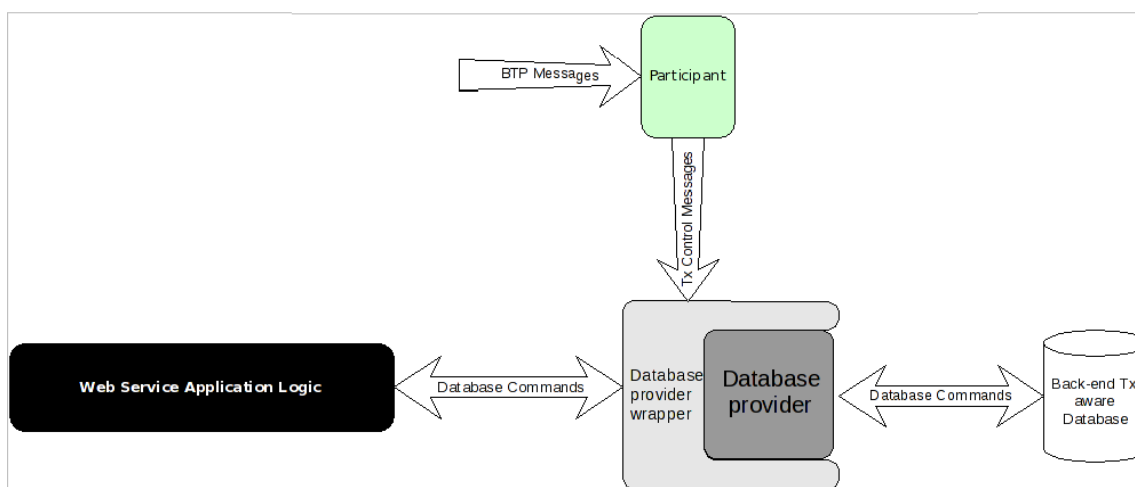

Figure 20.1. Transactions, Participants, and Back-End Transaction Control

20.1.1. Atomic Transaction

All Atomic Transaction participants are instances of the Section 20.1.1.1, "Durable2PCParticipant" or Section 20.1.1.2, "Volatile2PCParticipant".

20.1.1.1. Durable2PCParticipant

A Durable2PCParticipant supports the WS-Atomic Transaction Durable2PC protocol with the signatures listed in Durable2PCParticipant Signatures, as per the `com.arjuna.wst11.Durable2Participant` interface.

Durable2PCParticipant Signatures

`prepare`

> The participant should perform any work necessary, so that it can either commit or roll back the work performed by the Web service under the scope of the transaction. The implementation is free to do whatever it needs to in order to fulfill the implicit contract between it and the coordinator.
>
> The participant indicates whether it can `prepare` by returning an instance of the `com.arjuna.wst11.Vote`, with one of three values.

- **ReadOnly** indicates that the participant does not need to be informed of the transaction outcome, because it did not update any state information.

- **Prepared** indicates that the participant is ready to commit or roll back, depending on the final transaction outcome. Sufficient state updates have been made persistent to accomplish this.

- **Aborted** indicates that the participant has aborted and the transaction should also attempt to do so.

commit

The participant should make its work permanent. How it accomplishes this depends upon its implementation. For instance, in the theater example, the reservation of the ticket is committed. If commit processing cannot complete, the participant should throw a **SystemException** error, potentially leading to a heuristic outcome for the transaction.

rollback

The participant should undo its work. If rollback processing cannot complete, the participant should throw a **SystemException** error, potentially leading to a heuristic outcome for the transaction.

unknown

This method has been deprecated and is slated to be removed from XTS in the future.

error

In rare cases when recovering from a system crash, it may be impossible to complete or roll back a previously prepared participant, causing the **error** operation to be invoked.

20.1.1.2. Volatile2PCParticipant

This participant supports the WS-Atomic Transaction Volatile2PC protocol with the signatures listed in Volatile2PCParticipant Signatures, as per the **com.arjuna.wst11.Volatile2Participant** interface.

Volatile2PCParticipant Signatures

prepare

The participant should perform any work necessary to flush any volatile data created by the Web service under the scope of the transaction, to the system store. The implementation is free to do whatever it needs to in order to fulfill the implicit contract between it and the coordinator.

The participant indicates whether it can **prepare** by returning an instance of the **com.arjuna.wst11.Vote**, with one of three values.

- **ReadOnly** indicates that the participant does not need to be informed of the transaction outcome, because it did not change any state information during the life of the transaction.

- **Prepared** indicates that the participant wants to be notified of the final transaction outcome via a call to **commit** or **rollback**.

- **Aborted** indicates that the participant has aborted and the transaction should also attempt to do so.

commit

The participant should perform any cleanup activities required, in response to a successful transaction commit. These cleanup activities depend upon its implementation. For instance, it may flush cached backup copies of data modified during the transaction. In the unlikely event that commit processing cannot complete, the participant should throw a `SystemException` error. This will not affect the outcome of the transaction but will cause an error to be logged. This method may not be called if a crash occurs during commit processing.

rollback

The participant should perform any cleanup activities required, in response to a transaction abort. In the unlikely event that rollback processing cannot complete, the participant should throw a `SystemException` error. This will not affect the outcome of the transaction but will cause an error to be logged. This method may not be called if a crash occurs during commit processing.

unknown

This method is deprecated and will be removed in a future release of XTS.

error

This method should never be called, since volatile participants are not involved in recovery processing.

20.1.2. Business Activity

All Business Activity participants are instances one or the other of the interfaces described in Section 20.1.2.1, "BusinessAgreementWithParticipantCompletion" or Section 20.1.2.2, "BusinessAgreementWithCoordinatorCompletion" interface.

20.1.2.1. BusinessAgreementWithParticipantCompletion

The `BusinessAgreementWithParticipantCompletion` interface supports the WS-Transactions `BusinessAgreementWithParticipantCompletion` protocol with the signatures listed in BusinessAgreementWithParticipantCompletion Signatures, as per interface `com.arjuna.wst11.BusinessAgreementWithParticipantCompletionParticipant`.

BusinessAgreementWithParticipantCompletion Signatures

close

The transaction has completed successfully. The participant has previously informed the coordinator that it was ready to complete.

cancel

The transaction has canceled, and the participant should undo any work. The participant cannot have informed the coordinator that it has completed.

compensate

The transaction has canceled. The participant previously informed the coordinator that it had finished work but could compensate later if required, and it is now requested to do so. If compensation cannot be performed, the participant should throw a `FaultedException` error, potentially leading to a heuristic outcome for the transaction. If compensation processing cannot

complete because of a transient condition then the participant should throw a `SystemException` error, in which case the compensation action may be retried or the transaction may finish with a heuristic outcome.

status

Return the status of the participant.

unknown

This method is deprecated and will be removed a future XTS release.

error

In rare cases when recovering from a system crash, it may be impossible to compensate a previously-completed participant. In such cases the `error` operation is invoked.

20.1.2.2. BusinessAgreementWithCoordinatorCompletion

The BusinessAgreementWithCoordinatorCompletion participant supports the WS-Transactions `BusinessAgreementWithCoordinatorCompletion` protocol with the signatures listed in BusinessAgreementWithCoordinatorCompletion Signatures, as per the `com.arjuna.wst11.BusinessAgreementWithCoordinatorCompletionParticipant` interface.

BusinessAgreementWithCoordinatorCompletion Signatures

close

The transaction completed successfully. The participant previously informed the coordinator that it was ready to complete.

cancel

The transaction canceled, and the participant should undo any work.

compensate

The transaction canceled. The participant previously informed the coordinator that it had finished work but could compensate later if required, and it is now requested to do so. In the unlikely event that compensation cannot be performed the participant should throw a `FaultedException` error, potentially leading to a heuristic outcome for the transaction. If compensation processing cannot complete because of a transient condition, the participant should throw a `SystemException` error, in which case the compensation action may be retried or the transaction may finish with a heuristic outcome.

complete

The coordinator is informing the participant all work it needs to do within the scope of this business activity has been completed and that it should make permanent any provisional changes it has made.

status

Returns the status of the participant.

unknown

This method is deprecated and will be removed in a future release of XTS.

error

> In rare cases when recovering from a system crash, it may be impossible to compensate a previously completed participant. In such cases, the `error` method is invoked.

20.1.2.3. BAParticipantManager

In order for the Business Activity protocol to work correctly, the participants must be able to autonomously notify the coordinator about changes in their status. Unlike the Atomic Transaction protocol, where all interactions between the coordinator and participants are instigated by the coordinator when the transaction terminates, the BAParticipantManager interaction pattern requires the participant to be able to talk to the coordinator at any time during the lifetime of the business activity.

Whenever a participant is registered with a business activity, it receives a handle on the coordinator. This handle is an instance of interface com.arjuna.wst11.BAParticipantManager with the methods listed in BAParticipantManager Methods.

BAParticipantManager Methods

exit

> The participant uses the method `exit` to inform the coordinator that it has left the activity. It will not be informed when and how the business activity terminates. This method may only be invoked while the participant is in the `active` state (or the `completing` state, in the case of a participant registered for the `ParticipantCompletion` protocol). If it is called when the participant is in any other state, a `WrongStateException` error is thrown. An `exit` does not stop the activity as a whole from subsequently being closed or canceled/compensated, but only ensures that the exited participant is no longer involved in completion, close or compensation of the activity.

completed

> The participant has completed its work, but wishes to continue in the business activity, so that it will eventually be informed when, and how, the activity terminates. The participant may later be asked to compensate for the work it has done or learn that the activity has been closed.

fault

> The participant encountered an error during normal activation and has done whatever it can to compensate the activity. The `fault` method places the business activity into a mandatory `cancel-only` mode. The faulted participant is no longer involved in completion, close or compensation of the activity.

20.2. PARTICIPANT CREATION AND DEPLOYMENT

The participant provides the plumbing that drives the transactional aspects of the service. This section discusses the specifics of Participant programming and usage.

20.2.1. Implementing Participants

Implementing a participant is a relatively straightforward task. However, depending on the complexity of the transactional infrastructure that the participant needs to manage, the task can vary greatly in

complexity and scope. Your implementation needs to implement one of the interfaces found under com.arjuna.wst11.
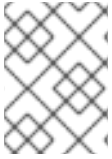
> **NOTE**
>
> The corresponding participant interfaces used in the 1.0 protocol implementation are located in package `com.arjuna.wst`.

20.2.2. Deploying Participants

Transactional web services and transactional clients are deployed by placing them in the application server deploy directory alongside the XTS service archive (SAR). The SAR exports all the client and web service API classes needed to manage transactions and enroll and manage participant web services. It provides implementations of all the WS-C and WS-T coordination services, not just the coordinator services. In particular, it exposes the client and web service participant endpoints which are needed to receive incoming messages originating from the coordinator.

Normally, a transactional application client and the transaction web service it invokes will be deployed in different application servers. As long as the XTS SAR is deployed to each of these containers XTS will transparently route coordination messages from clients or web services to their coordinator and vice versa. When the the client begins a transaction by default it creates a context using the coordination services in its local container. The context holds a reference to the local Registration Service which means that any web services enlisted in the transaction enrol with the coordination services in the same container."

The coordinator does not need to reside in the same container as the client application. By configuring the client deployment appropriately it is possible to use the coordinator services co-located with one of the web services or even to use services deployed in a separate, dedicated container. See Chapter 8 Stand-Alone Coordination for details of how to configure a coordinator located in a different container to the client.

> **WARNING**
>
> In previous releases, XTS applications were deployed using the appropriate XTS and Transaction Manager `.jar`, `.war`, and configuration files bundled with the application. This deployment method is no longer supported in the Enterprise Application Platform.

CHAPTER 21. THE XTS API

This chapter discusses the XTS API. You can use this information to write client and server applications which consume transactional Web Services and coordinate back-end systems.

21.1. API FOR THE ATOMIC TRANSACTION PROTOCOL

21.1.1. Vote

During the two-phase commit protocol, a participant is asked to vote on whether it can prepare to confirm the work that it controls. It must return an instance of one of the subtypes of `com.arjuna.wst11.Vote` listed in Subclasses of `com.arjuna.wst11.Vote`.

Subclasses of `com.arjuna.wst11.Vote`

Prepared

Indicates that the participant can prepare if the coordinator requests it. Nothing has been committed, because the participant does not know the final outcome of the transaction.

Aborted

The participant cannot prepare, and has rolled back. The participant should not expect to get a second phase message.

ReadOnly

The participant has not made any changes to state, and it does not need to know the final outcome of the transaction. Essentially the participant is resigning from the transaction.

Example 21.1. Example Implementation of 2PC Participant's `prepare` Method

```
public Vote prepare () throws WrongStateException, SystemException
{
  // Some participant logic here

  if(/* some condition based on the outcome of the business logic */)
  {
    // Vote to confirm
    return new com.arjuna.wst.Prepared();
  }
  else if(/*another condition based on the outcome of the business
logic*/)
  {
    // Resign
    return new com.arjuna.wst.ReadOnly();
  }
  else
  {
    // Vote to cancel
    return new com.arjuna.wst.Aborted();
  }
}
```

21.1.2. TXContext

com.arjuna.mw.wst11.TxContext is an opaque representation of a transaction context. It returns one of two possible values, as listed in .

TxContext Return Values

valid

Indicates whether the contents are valid.

equals

Can be used to compare two instances for equality.

> **NOTE**
>
> The corresponding participant interfaces used in the 1.0 protocol implementation are located in package com.arjuna.wst.

21.1.3. UserTransaction

`com.arjuna.mw.wst11.UserTransaction` is the class that clients typically employ. Before a client can begin a new atomic transaction, it must first obtain a `UserTransaction` from the `UserTransactionFactory`. This class isolates the user from the underlying protocol-specific aspects of the XTS implementation. A `UserTransaction` does not represent a specific transaction. Instead, it provides access to an implicit per-thread transaction context, similar to the `UserTransaction` in the JTA specification. All of the `UserTransaction` methods implicitly act on the current thread of control.

userTransaction Methods

begin

Used to begin a new transaction and associate it with the invoking thread.

Parameters

timeout

This optional parameter, measured in milliseconds, specifies a time interval after which the newly created transaction may be automatically rolled back by the coordinator

Exceptions

WrongStateException

A transaction is already associated with the thread.

commit

Volatile2PC and Durable2PC participants enrolled in the transaction are requested first to prepare and then to commit their changes. If any of the participants fails to prepare in the first phase then all other participants are requested to abort.

Exceptions

UnknownTransactionException

No transaction is associated with the invoking thread.

TransactionRolledBackException

The transaction was rolled back either because of a timeout or because a participant was unable to commit.

rollback

Terminates the transaction. Upon completion, the **rollback** method disassociates the transaction from the current leaving it unassociated with any transactions.

Exceptions

UnknownTransactionException

No transaction is associated with the invoking thread.

21.1.4. UserTransactionFactory

Call the **getUserTransaction** method to obtain a Section 21.1.3, "UserTransaction" instance from a **UserTransactionFactory**.

21.1.5. TransactionManager

Defines the interaction between a transactional web service and the underlying transaction service implementation. A **TransactionManager** does not represent a specific transaction. Instead, it provides access to an implicit per-thread transaction context.

Methods

currentTransaction

Returns a **TxContext** for the current transaction, or null if there is no context. Use the **currentTransaction** method to determine whether a web service has been invoked from within an existing transaction. You can also use the returned value to enable multiple threads to execute within the scope of the same transaction. Calling the **currentTransaction** method does not disassociate the current thread from the transaction.

suspend

Dissociates a thread from any transaction. This enables a thread to do work that is not associated with a specific transaction.

The **suspend** method returns a **TxContext** instance, which is a handle on the transaction.

resume

Associates or re-associates a thread with a transaction, using its **TxContext**. Prior to association or re-association, the thread is disassociated from any transaction with which it may be currently associated. If the **TxContext** is null, then the thread is associated with no transaction. In this way,

the result is the same as if the **suspend** method were used instead.

Parameters

txContext

A TxContext instance as return by **suspend,** identifying the transaction to be resumed.

Exceptions

UnknownTransactionException

The transaction referred to by the **TxContext** is invalid in the scope of the invoking thread.

enlistForVolitaleTwoPhase

Enroll the specified participant with the current transaction, causing it to participate in the Volatile2PC protocol. You must pass a unique identifier for the participant.

Parameters

participant

An implementation of interface Volatile2PCParticipant whose prepare, commit and abort methods are called when the corresponding coordinator message is received.

id

A unique identifier for the participant. The value of this String should differ for each enlisted participant. It should also be possible for a given identifier to determine that the participant belongs to the enlisting web service rather than some other web service deployed to the same container.

Exceptions

UnknownTransactionException

No transaction is associated with the invoking thread.

WrongStateException

The transaction is not in a state that allows participants to be enrolled. For instance, it may be in the process of terminating.

enlistForDurableTwoPhase

Enroll the specified participant with the current transaction, causing it to participate in the Durable2PC protocol. You must pass a unique identifier for the participant.

Exceptions

UnknownTransactionException

No transaction is associated with the invoking thread.

WrongStateException

> The transaction is not in a state that allows participants to be enrolled. For instance, it may be in the process of terminating.

21.1.6. TransactionManagerFactory

Use the **getTransactionManager** method to obtain a Section 21.1.5, "TransactionManager" from a **TransactionManagerFactory**.

21.2. API FOR THE BUSINESS ACTIVITY PROTOCOL

21.2.1. Compatibility

Previous implementations of XTS locate the Business Activity Protocol classes in the com.arjuna.mw.wst package. In the current implementation, these classes are located in the com.arjuna.mw.wst11 package.

21.2.2. UserBusinessActivity

com.arjuna.wst11.UserBusinessActivity is the class that most clients employ. A client begins a new business activity by first obtaining a **UserBusinessActivity** from the **UserBusinessActivityFactory**. This class isolates them from the underlying protocol-specific aspects of the XTS implementation. A UserBusinessActivity does not represent a specific business activity. Instead, it provides access to an implicit per-thread activity. Therefore, all of the **UserBusinessActivity** methods implicitly act on the current thread of control.

Methods

begin

> Begins a new activity, associating it with the invoking thread.

> **Parameters**

> **timeout**

> > The interval, in milliseconds, after which an activity times out. Optional.

> **Exceptions**

> **WrongStateException**

> > The thread is already associated with a business activity.

close

> First, all Coordinator Completion participants enlisted in the activity are requested to complete the activity. Next all participants, whether they enlisted for Coordinator or Participant Completion, are requested to close the activity. If any of the Coordinator Completion participants fails to complete at the first stage then all completed participants are asked to compensate the activity while any remaining uncompleted participants are requested to cancel the activity.

Exceptions

UnknownTransactionException

No activity is associated with the invoking thread.

TransactionRolledBackException

The activity has been cancelled because one of the Coordinator Completion participants failed to complete. This exception may also thrown if one of the Participant Completion participants has not completed before the client calls close.

cancel

Terminates the business activity. All Participant Completion participants enlisted in the activity which have already completed are requested to compensate the activity. All uncompleted Participant Completion participants and all Coordinator Completion participants are requested to cancel the activity.

Exceptions

UnknownTransactionException

No activity is associated with the invoking thread. Any participants that previous completed are directed to compensate their work.

21.2.3. UserBusinessActivityFactory

Use the `getuserbusinessActivity` method to obtain a [Section 21.2.2, "UserBusinessActivity"](#) instance from a `userBusinessActivityFactory`.

21.2.4. BusinessActivityManager

com.arjuna.mw.wst11.BusinessActivityManager is the class that web services typically employ. Defines how a web service interacts with the underlying business activity service implementation. A `BusinessActivityManager` does not represent a specific activity. Instead, it provides access to an implicit per-thread activity.

Methods

currentTransaction

Returns the `TxContext` for the current business activity, or **NULL** if there is no `TxContext`. The returned value can be used to enable multiple threads to execute within the scope of the same business activity. Calling the `currenTransaction` method does not dissociate the current thread from its activity.

suspend

Dissociates a thread from any current business activity, so that it can perform work not associated with a specific activity. The **suspend** method returns a `TxContext` instance, which is a handle on the activity. The thread is then no longer associated with any activity.

resume

Associates or re-associates a thread with a business activity, using its **TxContext**. Before associating or re-associating the thread, it is disassociated from any business activity with which it is currently associated. If the **TxContext** is **NULL**, the thread is disassociated with all business activities, as though the **suspend** method were called.

Parameters

txContext

> A TxContext instance as returned by **suspend**, identifying the transaction to be resumed.

Exceptions

UnknownTransactionException

> The business activity to which the **TxContext** refers is invalid in the scope of the invoking thread.

enlistForBusinessAgreementWithParticipantCompletion

Enroll the specified participant with current business activity, causing it to participate in the **BusinessAgreementWithParticipantCompletion** protocol. A unique identifier for the participant is also required.

The return value is an instance of BAParticipantManager which can be used to notify the coordinator of changes in the participant state. In particular, since the participant is enlisted for the Participant Completion protcol it is expected to call the completed method of this returned instance when it has completed all the work it expects to do in this activity and has made all its changes permanent. Alternatively, if the participant does not need to perform any compensation actions should some other participant fail it can leave the activity by calling the exit method of the returned BAParticipantManager instance.

Parameters

participant

> An implementation of interface **BusinessAgreementWithParticipantCompletionParticipant** whose **close**, **cancel**, and **compensate** methods are called when the corresponding coordinator message is received.

id

> A unique identifier for the participant. The value of this String should differ for each enlisted participant. It should also be possible for a given identifier to determine that the participant belongs to the enlisting web service rather than some other web service deployed to the same container.

Exceptions

UnknownTransactionException

> No transaction is associated with the invoking thread.

WrongStateException

The transaction is not in a state where new participants may be enrolled, as when it is terminating.

enlistForBusinessAgreementWithCoordinatorCompletion

Enroll the specified participant with current activity, causing it to participate in the **BusinessAgreementWithCoordinatorCompletion** protocol. A unique identifier for the participant is also required.

The return value is an instance of **BAParticipantManager** which can be used to notify the coordinator of changes in the participant state. Note that in this case it is an error to call the **completed** method of this returned instance. With the Coordinator Completion protocol the participant is expected to wait until its **completed** method is called before it makes all its changes permanent. Alternatively, if the participant determiens that it has no changes to make, it can leave the activity by calling the **exit** method of the returned **BAParticipantManager** instance.

Parameters

participant

An implementation of interface BusinessAgreementWithCoordinatorCompletionParticipant whose completed, close, cancel and compensate methods are called when the corresponding coordinator message is received.

id

A unique identifier for the participant. The value of this String should differ for each enlisted participant. It should also be possible for a given identifier to determine that the participant belongs to the enlisting web service rather than some other web service deployed to the same container.

Exceptions

>UnknownTransactionException

No transaction is associated with the invoking thread.

WrongStateException

The transaction is not in a state where new participants may be enrolled, as when it is terminating.

21.2.5. BusinessActivityManagerFactory

Use the **getBusinessActivityManager** method to obtain a Section 21.2.4, "BusinessActivityManager" instance from a **BusinessActivityManagerFactory**.

CHAPTER 22. STAND-ALONE COORDINATION

22.1. INTRODUCTION

The XTS service is deployed as a JBoss service archive (SAR). The version of the service archive provided with the Transaction Service implements version 1.1 of the WS-C, WS-AT and WS-BA services. You can rebuild the XTS service archive to include both the 1.0 and 1.1 implementations and deploy them side by side. See the service archive build script for for further details.

The release service archive obtains coordination contexts from the Activation Coordinator service running on the deployed host. Therefore, WS-AT transactions or WS-BA activities created by a locally-deployed client application are supplied with a context which identifies the Registration Service running on the client's machine. Any Web Services invoked by the client are coordinated by the Transaction Protocol services running on the client's host. This is the case whether the Web Services are running locally or remotely. Such a configuration is called *local coordination*.

You can reconfigure this setting globally for all clients, causing context creation requests to be redirected to an Activation Coordinator Service running on a remote host. Normally, the rest of the coordination process is executed from the remote host. This configuration is called *stand-alone coordination*.

Reasons for Choosing a Stand-Alone Coordinator

- Efficiency: if a client application invokes Web Services on a remote Enterprise Application Platform server, coordinating the transaction from the remote server might be more efficient, since the protocol-specific messages between the coordinator and the participants do not need to travel over the network.

- Reliability: if the coordinator service runs on a dedicated host, there is no danger of failing applications or services affecting the coordinator and causing failures for unrelated transactions.

- A third reason might be to use a coordination service provided by a third party vendor.

22.2. CONFIGURING THE ACTIVATION COORDINATOR

The simplest way to configure a stand-alone coordinator is to provide a command line switch when starting the Enterprise Application Platform. The **-D** option specifies a setting for a System property. Several configuration options can be enabled, taking effect in Section 22.2.1, "Command-Line Options Passed with the **-D** Parameter, Ordered by Priority" .

22.2.1. Command-Line Options Passed with the **-D** Parameter, Ordered by Priority

Absolute URL

- Property: `org.jboss.jbossts.xts11.coordinatorURL`

- Format: http://coord.host:coord.port/ws-c11/ActivationService

- The value assigned to these URLs depends upon the configuration of the remote coordinator host. The sample values listed with the property names are appropriate when the coordinator is another JBoss Transaction Service XTS service. Substitute the *coord.host* and *coord.port* with the appropriate values for the Enterprise Application Platform instance running the Activation Coordinator service.

Coordinator Host Information

- **Property**

  - org.jboss.jbossts.xts11.coordinator.host

  - org.jboss.jbossts.xts11.coordinator.port

  - org.jboss.jbossts.xts11.coordinator.path

  Format

  - server.bind.address

  - jboss.web.bind.port

  - ws-c11/ActivationService

- If you set any of these three components, the coordinator URL is constructed using whichever of the component values is defined and substituting the default values specified for any undefined components. The values *server.bind.address* and *jboss.web.bind.port* represent the server bind address and the web service listener port obtained either from the application server command-line or the server configuration files.

CHAPTER 23. PARTICIPANT CRASH RECOVERY

A key requirement of a transaction service is to be resilient to a system crash by a host running a participant, as well as the host running the transaction coordination services. Crashes which happen before a transaction terminates or before a business activity completes are relatively easy to accommodate. The transaction service and participants can adopt a *presumed abort* policy.

Procedure 23.1. Presumed Abort Policy

1. If the coordinator crashes, it can assume that any transaction it does not know about is invalid, and reject a participant request which refers to such a transaction.

2. If the participant crashes, it can forget any provisional changes it has made, and reject any request from the coordinator service to prepare a transaction or complete a business activity.

Crash recovery is more complex if the crash happens during a transaction commit operation, or between completing and closing a business activity. The transaction service must ensure as far as possible that participants arrive at a consistent outcome for the transaction.

WS-AT Transaction

The transaction needs to commit all provisional changes or roll them all back to the state before the transaction started.

WS-Business Activity Transaction

All participants need to close the activity or cancel the activity, and run any required compensating actions.

On the rare occasions where such a consensus cannot be reached, the transaction service must log and report transaction failures.

XTS includes support for automatic recovery of WS-AT and WS-BA transactions, if either or both of the coordinator and participant hosts crashes. The XTS recovery manager begins execution on coordinator and participant hosts when the XTS service restarts. On a coordinator host, the recovery manager detects any WS-AT transactions which have prepared but not committed, as well as any WS-BA transactions which have completed but not yet closed. It ensures that all their participants are rolled forward in the first case, or closed in the second.

On a participant host, the recovery manager detects any prepared WS-AT participants which have not responded to a transaction rollback, and any completed WS-BA participants which have not yet responded to an activity cancel request, and ensures that the former are rolled back and the latter are compensated. The recovery service also allows for recovery of subordinate WS-AT transactions and their participants if a crash occurs on a host where an interposed WS-AT coordinator has been employed.

23.1. WS-AT RECOVERY

23.1.1. WS-AT Coordinator Crash Recovery

The WS-AT coordination service tracks the status of each participant in a transaction as the transaction progresses through its two-phase commit. When all participants have been sent a **prepare** message and have responded with a **prepared** message, the coordinator writes a log record storing each participant's details, indicating that the transaction is ready to complete. If the coordinator service crashes after this point has been reached, completion of the two-phase commit protocol is still

guaranteed, by reading the log file after reboot and sending a `commit` message to each participant. Once all participants have responded to the `commit` with a `committed` message, the coordinator can safely delete the log entry.

Since the `prepared` messages returned by the participants imply that they are ready to commit their provisional changes and make them permanent, this type of recovery is safe. Additionally, the coordinator does not need to account for any commit messages which may have been sent before the crash, or resend messages if it crashes several times. The XTS participant implementation is resilient to redelivery of the `commit` messages. If the participant has implemented the recovery functions described in Section 23.1.2.1, "WS-AT Participant Crash Recovery APIs", the coordinator can guarantee delivery of `commit` messages if both it crashes, and one or more of the participant service hosts also crash, at the same time.

If the coordination service crashes before the `prepare` phase completes, the presumed abort protocol ensures that participants are rolled back. After system restart, the coordination service has the information about about all the transactions which could have entered the `commit` phase before the reboot, since they have entries in the log. It also knows about any active transactions started after the reboot. If a participant is waiting for a response, after sending its `prepared` message, it automatically re sends the `prepared` message at regular intervals. When the coordinator detects a transaction which is not active and has no entry in the log file after the reboot, it instructs the participant to abort, ensuring that the web service gets a chance to roll back any provisional state changes it made on behalf of the transaction.

A web service may decide to unilaterally commit or roll back provisional changes associated with a given participant, if configured to time out after a specified length of time without a response. In this situation, the the web service should record this action and log a message to persistent storage. When the participant receives a request to commit or roll back, it should throw an exception if its unilateral decision action does not match the requested action. The coordinator detects the exception and logs a message marking the outcome as heuristic. It also saves the state of the transaction permanently in the transaction log, to be inspected and reconciled by an administrator.

23.1.2. WS-AT Participant Crash Recovery

WS-AT participants associated with a transactional web service do not need to be involved in crash recovery if the Web service's host machine crashes before the participant is told to prepare. The coordinator will assume that the transaction has aborted, and the Web service can discard any information associated with unprepared transactions when it reboots.

When a participant is told to `prepare`, the Web service is expected to save to persistent storage the transactional state it needs to commit or roll back the transaction. The specific information it needs to save is dependent on the implementation and business logic of the Web Service. However, the participant must save this state before returning a `Prepared` vote from the `prepare` call. If the participant cannot save the required state, or there is some other problem servicing the request made by the client, it must return an `Aborted` vote.

The XTS participant services running on a Web Service's host machine cooperate with the Web service implementation to facilitate participant crash recovery. These participant services are responsible for calling the participant's `prepare`, `commit`, and `rollback` methods. The XTS implementation tracks the local state of every enlisted participant. If the `prepare` call returns a `Prepared` vote, the XTS implementation ensures that the participant state is logged to the local transaction log before forwarding a `prepared` message to the coordinator.

A participant log record contains information identifying the participant, its transaction, and its coordinator. This is enough information to allow the rebooted XTS implementation to reinstate the participant as active and to continue communication with the coordinator, as though the participant

had been enlisted and driven to the prepared state. However, a participant instance is still necessary for the commit or rollback process to continue.

Full recovery requires the log record to contain information needed by the Web service which enlisted the participant. This information must allow it to recreate an equivalent participant instance, which can continue the `commit` process to completion, or roll it back if some other Web Service fails to `prepare`. This information might be as simple as a String key which the participant can use to locate the data it made persistent before returning its Prepared vote. It may be as complex as a serialized object tree containing the original participant instance and other objects created by the Web service.

If a participant instance implements the relevant interface, the XTS implementation will append this participant recovery state to its log record before writing it to persistent storage. In the event of a crash, the participant recovery state is retrieved from the log and passed to the Web Service which created it. The Web Service uses this state to create a new participant, which the XTS implementation uses to drive the transaction to completion. Log records are only deleted after the participant's `commit` or `rollback` method is called.

> ⚠️ **WARNING**
>
> If a crash happens just before or just after a `commit` method is called, a `commit` or `rollback` method may be called twice.

23.1.2.1. WS-AT Participant Crash Recovery APIs

23.1.2.1.1. Saving Participant Recovery State

To signal that it is capable of performing recovery processing, a participant can implement the `java.lang.Serializable` interface. Alternatively it may implement Example 23.1, "The PersistableATParticipant Interface".

Example 23.1. The PersistableATParticipant Interface

```
public interface PersistableATParticipant
{
  byte[] getRecoveryState() throws Exception;
}
```

If a participant implements the `Serializable` interface, the XTS participant services implementation uses the serialization API to create a version of the participant which can be appended to the participant log entry. If it implements the `PersistableATParticipant` interface, the XTS participant services implementation call the `getRecoveryState` method to obtain the state to be appended to the participant log entry.

If neither of these APIs is implemented, the XTS implementation logs a warning message and proceeds without saving any recovery state. In the event of a crash on the host machine for the Web service during commit, the transaction cannot be recovered and a heuristic outcome may occur. This outcome is logged on the host running the coordinator services.

23.1.2.1.2. Recovering Participants at Reboot

A Web service must register with the XTS implementation when it is deployed, and unregister when it is undeployed, in order to participate in recovery processing. Registration is performed using class **XTSATRecoveryManager** defined in package org.jboss.jbossts.xts.recovery.participant.at.

Example 23.2. Registering for Recovery

```
public abstract class XTSATRecoveryManager {
  . . .
   public static XTSATRecoveryManager getRecoveryManager() ;
   public void registerRecoveryModule(XTSATRecoveryModule module);
  public abstract void unregisterRecoveryModule(XTSATRecoveryModule
module)
   throws NoSuchElementException;
  . . .
}
```

The Web service must provide an implementation of the **XTSATRecoveryModule**, located in the org.jboss.jbossts.xts.recovery.participant.at, as argument to both the **register** and **unregister** calls. This instance is responsible for identifying saved participant recovery records and recreating new, recovered participant instances.

Example 23.3. XTSATRecoveryModule Implementation

```
public interface XTSATRecoveryModule
{
    public Durable2PCParticipant
 deserialize(String id, ObjectInputStream stream)
   throws Exception;
    public Durable2PCParticipant
 recreate(String id, byte[] recoveryState)
   throws Exception;
}
```

If a participant's recovery state was saved using serialization, the recovery module's **deserialize** method is called to recreate the participant. Normally, the recovery module is required to read, cast, and return an object from the supplied input stream. If a participant's recovery state was saved using the **PersistableATParticipant** interface, the recovery module's **recreate** method is called to recreate the participant from the byte array it provided when the state was saved.

The XTS implementation cannot identify which participants belong to which recovery modules. A module only needs to return a participant instance if the recovery state belongs to the module's Web service. If the participant was created by another Web service, the module should return **null**. The participant identifier, which is supplied as argument to the **deserialize** or **recreate** method, is the identifier used by the Web service when the original participant was enlisted in the transaction. Web Services participating in recovery processing should ensure that participant identifiers are unique per service. If a module recognizes that a participant identifier belongs to its Web service, but cannot recreate the participant, it should throw an exception. This situation might arise if the service cannot associate the participant with any transactional information which is specific to the business logic.

Even if a module relies on serialization to create the participant recovery state saved by the XTS

implementation, it still must be registered by the application. The `deserialization` operation must employ a class loader capable of loading classes specific to the Web service. XTS fulfills this requirement by devolving responsibility for the `deserialize` operation to the recovery module.

23.2. WS-BA RECOVERY

23.2.1. WS-BA Coordinator Crash Recovery

The WS-BA coordination service implementation tracks the status of each participant in an activity as the activity progresses through completion and closure. A transition point occurs during closure, once all `CoordinatorCompletion` participants receive a `complete` message and respond with a `completed` message. At this point, all `ParticipantCompletion` participants should have sent a `completed` message. The coordinator writes a log record storing the details of each participant, and indicating that the transaction is ready to close. If the coordinator service crashes after the log record is written, the `close` operation is still guaranteed to be successful. The coordinator checks the log after the system reboots and re sends a `close` message to all participants. After all participants respond to the `close` with a `closed` message, the coordinator can safely delete the log entry.

The coordinator does not need to account for any `close` messages sent before the crash, nor resend messages if it crashes several times. The XTS participant implementation is resilient to redelivery of `close` messages. Assuming that the participant has implemented the recovery functions described below, the coordinator can even guarantee delivery of `close` messages if both it, and one or more of the participant service hosts, crash simultaneously.

If the coordination service crashes before it has written the log record, it does not need to explicitly compensate any completed participants. The presumed abort protocol ensures that all completed participants are eventually sent a `compensate` message. Recovery must be initiated from the participant side.

A log record does not need to be written when an activity is being canceled. If a participant does not respond to a `cancel` or `compensate` request, the coordinator logs a warning and continues. The combination of the presumed abort protocol and participant-led recovery ensures that all participants eventually get canceled or compensated, as appropriate, even if the participant host crashes.

If a completed participant does not detect a response from its coordinator after resending its `completed` response a suitable number of times, it switches to sending `getstatus` messages, to determine whether the coordinator still knows about it. If a crash occurs before writing the log record, the coordinator has no record of the participant when the coordinator restarts, and the `getstatus` request returns a fault. The participant recovery manager automatically compensates the participant in this situation, just as if the activity had been canceled by the client.

After a participant crash, the participant recovery manager detects the log entries for each completed participant. It sends `getstatus` messages to each participant's coordinator host, to determine whether the activity still exists. If the coordinator has not crashed and the activity is still running, the participant switches back to resending `completed` messages, and waits for a `close` or `compensate` response. If the coordinator has also crashed or the activity has been canceled, the participant is automatically canceled.

23.2.2. WS-BA Participant Crash Recovery APIs

23.2.2.1. Saving Participant Recovery State

A participant may signal that it is capable of performing recovery processing, by implementing the **java.lang.Serializable** interface. An alternative is to implement the Example 23.4, "**PersistableBAParticipant** Interface".

Example 23.4. PersistableBAParticipant Interface

```
public interface PersistableBAParticipant
{
    byte[] getRecoveryState() throws Exception;
}
```

If a participant implements the **Serializable** interface, the XTS participant services implementation uses the serialization API to create a version of the participant which can be appended to the participant log entry. If the participant implements the **PersistableBAParticipant**, the XTS participant services implementation call the **getRecoveryState** method to obtain the state, which is appended to the participant log entry.

If neither of these APIs is implemented, the XTS implementation logs a warning message and proceeds without saving any recovery state. If the Web service's host machine crashes while the activity is being closed, the activity cannot be recovered and a heuristic outcome will probably be logged on the coordinator's host machine. If the activity is canceled, the participant is not compensated and the coordinator host machine may log a heuristic outcome for the activity.

23.2.2.2. Recovering Participants at Reboot

A Web service must register with the XTS implementation when it is deployed, and unregister when it is undeployed, so it can take part in recovery processing.

Registration is performed using the **XTSBARecoveryManager**, defined in the org.jboss.jbossts.xts.recovery.participant.ba package.

Example 23.5. XTSBARecoveryManager Class

```
public abstract class XTSBARecoveryManager {
    . . .
  public static XTSBARecoveryManager getRecoveryManager() ;
    public void registerRecoveryModule(XTSBARecoveryModule module);
    public abstract void unregisterRecoveryModule(XTSBARecoveryModule
module)
  throws NoSuchElementException;
    . . .
}
```

The Web service must provide an implementation of the **XTSBARecoveryModule** in the org.jboss.jbossts.xts.recovery.participant.ba, as an argument to the **register** and **unregister** calls. This instance identifies saved participant recovery records and recreates new, recovered participant instances:

Example 23.6. XTSBARecoveryModule Interface

```
public interface XTSATRecoveryModule
```

```
{
    public BusinessAgreementWithParticipantCompletionParticipant
  deserializeParticipantCompletionParticipant(String id,
          ObjectInputStream stream)
  throws Exception;
    public BusinessAgreementWithParticipantCompletionParticipant
  recreateParticipantCompletionParticipant(String id,
        byte[] recoveryState)
  throws Exception;
    public BusinessAgreementWithCoordinatorCompletionParticipant
  deserializeCoordinatorCompletionParticipant(String id,
          ObjectInputStream stream)
  throws Exception;
    public BusinessAgreementWithCoordinatorCompletionParticipant
  recreateCoordinatorCompletionParticipant(String id,
        byte[] recoveryState)
  throws Exception;
}
```

If a participant's recovery state was saved using serialization, one of the recovery module's **deserialize** methods is called, so that it can recreate the participant. Which method to use depends on whether the saved participant implemented the **ParticipantCompletion** protocol or the **CoordinatorCompletion** protocol. Normally, the recovery module reads, casts and returns an object from the supplied input stream. If a participant's recovery state was saved using the **PersistableBAParticipant** interface, one of the recovery module's **recreate** methods is called, so that it can recreate the participant from the byte array provided when the state was saved. The method to use depends on which protocol the saved participant implemented.

The XTS implementation does not track which participants belong to which recovery modules. A module is only expected to return a participant instance if it can identify that the recovery state belongs to its Web service. If the participant was created by some other Web service, the module should return **null**. The participant identifier supplied as an argument to the **deserialize** or **recreate** calls is the identifier used by the Web service when the original participant was enlisted in the transaction. Web Services which participate in recovery processing should ensure that the participant identifiers they employ are unique per service. If a module recognizes a participant identifier as belonging to its Web service, but cannot recreate the participant, it throws an exception. This situation might arise if the service cannot associate the participant with any transactional information specific to business logic.

A module must be registered by the application, even when it relies upon serialization to create the participant recovery state saved by the XTS implementation. The **deserialization** operation must employ a class loader capable of loading Web service-specific classes. The XTS implementation achieves this by delegating responsibility for the **deserialize** operation to the recovery module.

APPENDIX C. REVISION HISTORY

Revision 5.1.0-111.400   2013-10-31   **Rüdiger Landmann**
  Rebuild with publican 4.0.0

Revision 5.1.0-111   2012-07-18   **Anthony Towns**
  Rebuild for Publican 3.0

Revision 5.1.0-110   Wed Sep 15 2010   **Misty Stanley-Jones**
  Changed version number in line with new versioning requirements.
  Revised for JBoss Enterprise Application Platform 5.1.0.GA, including:
  Merged the JTA, JTS, and XTS API guides into one master guide.

INDEX

A

ACID, **Transactions Overview**

activation, **Overview of Protocols Used by XTS**

Activation Coordinator, **Overview of Protocols Used by XTS**

activation coordinator, **Stand-Alone Coordination**

active component, **Introduction**

API, **The XTS API**

atomic transactions

    atomicity, **Participants**

B

BAParticipantManager, **Participants**

business activities, **Getting Started**

BusinessActivityManager, **The XTS API**

BusinessActivityManagerFactory, **The XTS API**

BusinessAgreementWithCoordinatorCompletion, **Overview of Protocols Used by XTS** , **Participants**

BusinessAgreementWithParticipantCompletion, **Overview of Protocols Used by XTS** , **Participants**

C

com.arjuna.mw.wst11

    XTS API, **Getting Started**

command-line options, **Stand-Alone Coordination**

completion, **Overview of Protocols Used by XTS**

context handlers, **Getting Started**

coordination context, **Overview of Protocols Used by XTS**

Coordinator, **Transactions Overview**

D

DE

    Document Exchange, **Introduction**

deployment, **Participants**

Durable2PC, **Overview of Protocols Used by XTS**

Durable2PCParticipant, **Participants**

F

fault-tolerance, **Introduction**

H