ArjunaCore 4.15.1

ArjunaCore Development Guide

TxCore and TXOJ Programmers Guide

Mark Little

Jonathan Halliday

Andrew Dinn

Kevin Connor

ArjunaCore 4.15.1 ArjunaCore Development Guide
TxCore and TXOJ Programmers Guide
Edition 0

Author	Mark Little	*mlittle@redhat.com*
Author	Jonathan Halliday	*jhallida@redhat.com*
Author	Andrew Dinn	*adinn@redhat.com*
Author	Kevin Connor	*kconnor@redhat.com*
Editor	Misty Stanley-Jones	*misty@redhat.com*

This guide is most relevant to engineers who are responsible for administering JBoss Transactions installations. Although this guide is specifically intended for service developers, it will be useful to anyone who would like to gain an understanding of transactions and how they function.

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System**□□□ **Preferences**□□□ **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box

[1] https://fedorahosted.org/liberation-fonts/

and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications□□□ Accessories □□ Character Map** from the main menu bar. Next, choose **Search□□□ Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit□□ Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

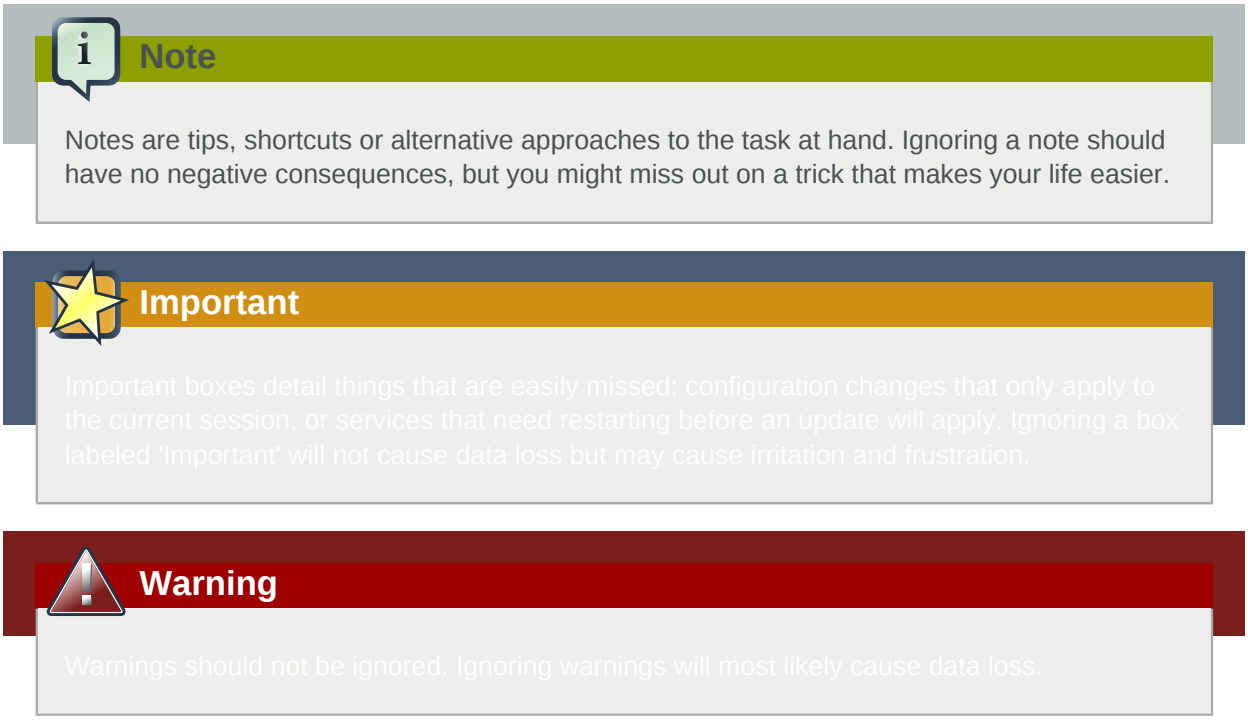
```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the JBoss Issue Tracker: *https://jira.jboss.org/* against the product **Documentation.**

When submitting a bug report, be sure to mention the manual's identifier: *ArjunaCore_Development_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

About This Guide

The TxCore and TXOJ Programmers Guide contains information on how to use Documentation. This document provides a detailed look at the design and operation of the TxCore transaction engine and the Transactional Objects for Java toolkit. It describes the architecture and the interaction of components within this architecture.

1.1. Audience

This guide is most relevant to engineers who want to use Documentation in installations that are not covered elsewhere. It is assumed that the reader is already familiar with the core Documentation documentation set.

1.2. Prerequisites

This guide assumes a basic familiarity with Java service development and object-oriented programming. A fundamental level of understanding in the following areas will also be useful:

- General understanding of the APIs, components, and objects that are present in Java applications.
- A general understanding of the Windows and UNIX operating systems.

Overview

A transaction is a unit of work that encapsulates multiple database actions such that that either all the encapsulated actions fail or all succeed.

Transactions ensure data integrity when an application interacts with multiple datasources.

This chapter contains a description of the use of the TxCore transaction engine and the **Transactional Objects for Java (TXOJ)** classes and facilities. The classes mentioned in this chapter are the key to writing fault-tolerant applications using transactions. Thus, they are described and then applied in the construction of a simple application. The classes to be described in this chapter can be found in the *com.arjuna.ats.txoj* and *com.arjuna.ats.arjuna* packages.

Stand-Alone Transaction Manager

Although JBoss Transaction Service can be embedded in various containers, such as JBoss Application Server, it remains a stand-alone transaction manager as well. There are no dependencies between the core JBoss Transaction Service and any container implementations.

2.1. TxCore

The Transaction Engine

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 need to be inherited, for example, concurrency control and state management. Other mechanisms, such as object storage and transactions, are implemented as TxCore objects that are created and manipulated like any other object.

Note

When the manual talks about using persistence and concurrency control facilities it assumes that the **Transactional Objects for Java (TXOJ)** classes are being used. If this is not the case then the programmer is responsible for all of these issues.

TxCore exploits object-oriented techniques to present programmers with a toolkit of Java classes from 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 in *Figure 2.1, “TxCore Class Hierarchy”* and which will be described later in this document.

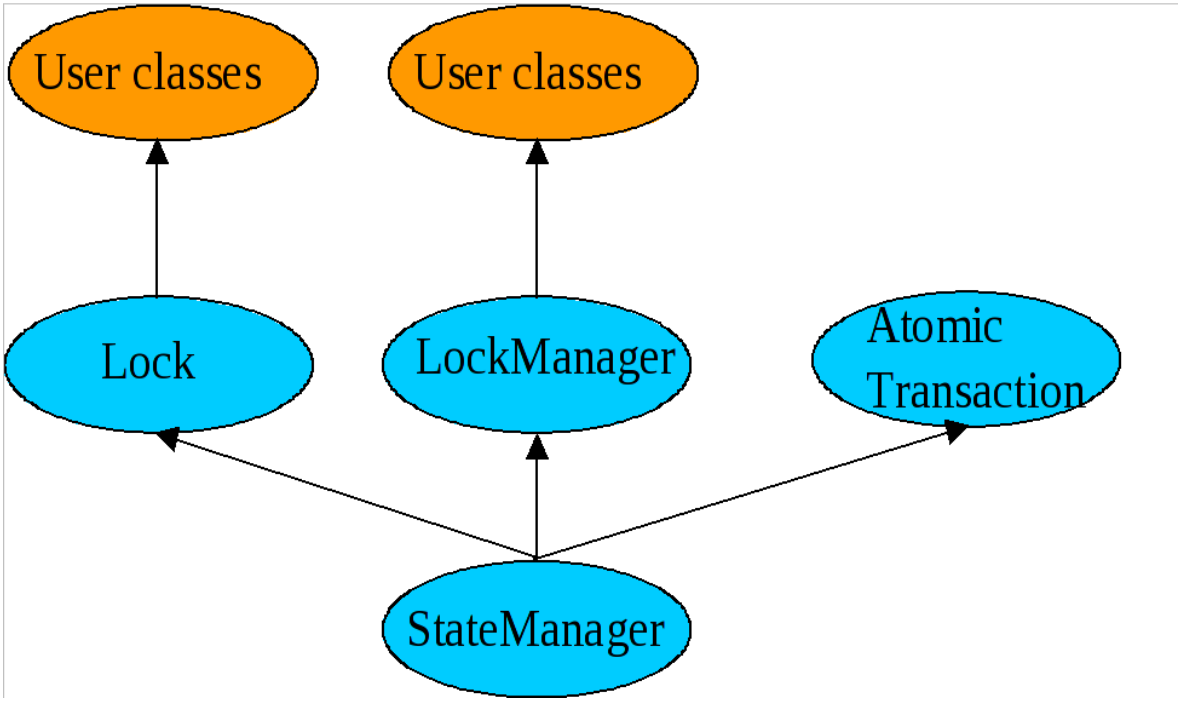


Figure 2.1. TxCore Class Hierarchy

Apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: TxCore and **TXOJ** guarantee that transactional objects will be registered with, and be driven by, the appropriate transactions, and crash recovery mechanisms are invoked automatically in the event of failures.

2.2. Saving object states

TxCore needs to be able to remember the state of an object for several purposes.

recovery
: The state represents some past state of the object.

persistence
: The state represents the final state of an object at application termination.

Since these requirements have common functionality they are all implemented using the same mechanism: the classes **`InputObjectState`** and **`OutputObjectState`**. The classes maintain an internal array into which instances of the standard types can be contiguously packed or unpacked using appropriate `pack` or `unpack` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form called *network byte order*, making them machine independent. Any other architecture-independent format, such as XDR or ASN.1, can be implemented simply by replacing the operations with ones appropriate to the encoding required.

2.3. The object store

Implementations of persistence can be affected by restrictions imposed by the Java SecurityManager. Therefore, the object store provided with TxCore is implemented using the techniques of interface and implementation. The current distribution includes implementations which write object states to the local file system or database, and remote implementations, where the interface uses a client stub (proxy) to remote services.

Persistent objects are assigned unique identifiers, which are instances of the **`Uid`** class, when they are created. These identifiers are used to identify them within the object store. States are read using the `read_committed` operation and written by the `write_committed` and `write_uncommitted` operations.

2.4. Recovery and persistence

At the root of the class hierarchy is the class **`StateManager`**. **`StateManager`** is responsible for object activation and deactivation, as well as object recovery. Refer to *Example 2.1, "**`Statemanager`**"* for the simplified signature of the class.

Example 2.1. **`Statemanager`**

```
 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 are assumed to be of three possible flavors.

Three Flavors of Objects

Recoverable
: **`StateManager`** attempts to generate and maintain appropriate recovery information for the object. Such objects have lifetimes that do not exceed the application program that creates them.

Recoverable and Persistent
: The lifetime of the object is assumed to be greater than that of the creating or accessing application, so that 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 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 operations `save_state` while performing `deactivate`, and `restore_state` while performing `activate`,) at various points during the execution of the application. These operations must be implemented by the programmer since **`StateManager`** cannot detect user-level state changes. This gives the programmer the ability to decide which parts of an object's state should be made persistent. For example, for a spreadsheet it may not be necessary to save all entries if some values can simply be recomputed. The `save_state` implementation for a class **`Example`** that has integer member variables called A, B and C might be implemented as in *Example 2.2, "`save_state` Implementation"*.

Example 2.2. `save_state` Implementation

```
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

it is necessary for all `save_state` and `restore_state` methods to call `super.save_state` and `super.restore_state`. This is to cater for improvements in the crash recovery mechanisms.

2.5. The life cycle of a Transactional Object for Java

A persistent object not in use is assumed to be held in a passive state, with its state residing in an object store and activated on demand. The fundamental life cycle of a persistent object in TXOJ is shown in *Figure 2.2, "Life cycle of a persistent Object in TXOJ"*.

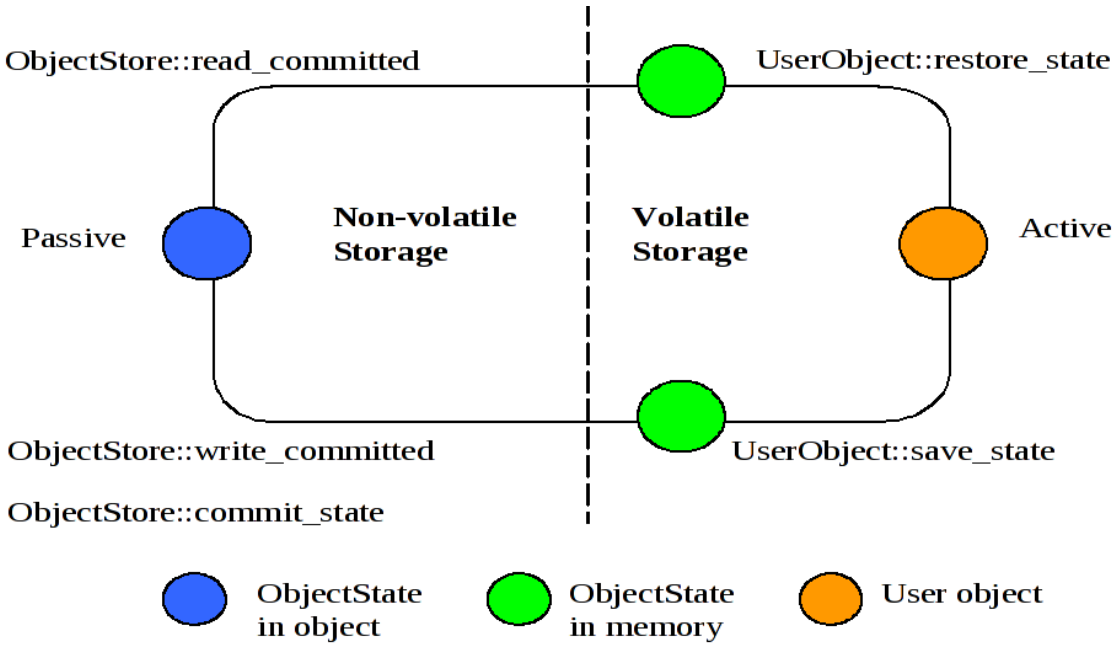


Figure 2.2. Life cycle of a persistent Object in TXOJ

Note

During its life time, a persistent object may be made active then passive many times.

2.6. The concurrency controller

The concurrency controller is implemented by the class **`LockManager`**, which provides sensible default behavior while allowing the programmer to override it if deemed necessary by the particular semantics of the class being programmed. As with **`StateManager`** and persistence, concurrency control implementations are accessed through interfaces. As well as providing access to remote services, the current implementations of concurrency control available to interfaces include:

Local disk/database implementation
: Locks are made persistent by being written to the local file system or database.

A purely local implementation
: Locks are maintained within the memory of the virtual machine which created them. This implementation has better performance than when 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 programmer interface 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 2.1, "TxCore Class Hierarchy"*, by inheriting from the **`Lock`** class, you can provide your own lock implementations with different lock conflict rules to enable type specific concurrency control.

Lock acquisition is, of necessity, under programmer control, since just as **`StateManager`** cannot determine if an operation modifies an object, **`LockManager`** cannot determine if an operation requires 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 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, int objectModel);
    protected LockManager (Uid storeUid);
    protected LockManager (Uid storeUid, int ot);
    protected LockManager (Uid storeUid, int ot, int objectModel);

    protected void terminate ();
};
```

The **`LockManager`** class is primarily responsible for managing requests to set a lock on an object or to release a lock as appropriate. However, since it is derived from **`StateManager`**, 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. This may in turn cause recovery information to be saved if the object is recoverable. In a similar fashion, successful lock acquisition causes `activate` to be invoked.

*Example 2.3, "**`Example`** Class"* shows how to try to obtain a write lock on an object.

Example 2.3. **`Example`** Class

```
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;
    }
}
```

2.7. The transactional 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, and, for those situations where programmers need to implement their own resources, methods for registering them with the current transaction. Because TxCore supports sub-transactions, if a transaction is begun within the scope of an already executing transaction it will automatically be nested.

You can use TxCore with multi-threaded applications. Each thread within an application can share a transaction or execute within its own transaction. Therefore, all TxCore classes are also thread-safe.

Example 2.4. 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 & objects deactivated */
}             /* (v) */
```

Creation of bindings to persistent objects

This could involve the creation of stub objects and a call to remote objects. Here, we re-bind 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.

Start of the atomic transaction

Operation invocations

As a part of a given invocation, the object implementation is responsible to ensure 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 acquired, if possible, from the object store.

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

Breaking of the previously created bindings

2.8. The class hierarchy

The principal classes which make up the class hierarchy of TxCore are depicted below.

- **StateManager**
  - **LockManager**
    - User-Defined Classes
  - **Lock**
    - User-Defined Classes
  - **AbstractRecord**
    - **RecoveryRecord**
    - **LockRecord**
    - **RecordList**
    - Other management record types
- **AtomicAction**
  - **TopLevelTransaction**
- **Input/OutputObjectBuffer**
  - **Input/OutputObjectState**
- **ObjectStore**

Programmers of fault-tolerant applications will be primarily concerned with the classes **LockManager**, **Lock**, and **AtomicAction**. Other classes important to a programmer are **Uid** and **ObjectState**.

Most TxCore 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 de-activation 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 concurrency control consists of two-phase locking in the current implementation. 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 serializability property requires that a write lock must be acquired on `O` before it is modified. Therefore, the body of `op1` should contain a call to the `setlock` operation of the concurrency controller.

Example 2.5. Simple Concurrency Control

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

Procedure 2.1. Steps followed by the operation `setlock`

The operation `setlock`, provided by the **`LockManager`** class, performs the following functions in *Example 2.5, "Simple Concurrency Control"*.

1. Check write lock compatibility with the currently held locks, and if allowed, continue.
2. Call the StateManager operation `activate`.`activate` will load, if not done already, the latest persistent state of **`O`** from the object store, then call the **`StateManager`** operation `modified`, which has the effect of creating an instance of either **`RecoveryRecord`** or **`PersistenceRecord`** for **`O`**, depending upon whether **`O`** was persistent or not. The Lock is a WRITE lock so the old state of the object must be retained prior to modification. The record is then inserted into the RecordList of Trans.
3. Create and insert a **`LockRecord`** instance in the **`RecordList`** of `Trans`.

Now suppose that action `Trans` is aborted sometime after the lock has been acquired. Then the `rollback` operation of **`AtomicAction`** will process 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 will release the WRITE lock while that of **`RecoveryRecord`** or **`PersistenceRecord`** will restore the prior state of **`O`**.

It is important to realize that all of the above work is automatically being performed by TxCore on behalf of the application programmer. The programmer need only start the transaction and set an appropriate lock; TxCore and **TXOJ** take care of participant registration, persistence, concurrency control and recovery.

Using TxCore

This section describes TxCore and **Transactional Objects for Java (TXOJ)** in more detail, and shows how to use TxCore to construct transactional applications.

3.1. State management

3.1.1. Object states

TxCore needs to be able to remember the state of an object for several purposes, including recovery (the state represents some past state of the object), and for persistence (the state represents the final state of an object at application termination). Since all of these requirements require common functionality they are all implemented using the same mechanism - the classes Input/OutputObjectState and Input/OutputBuffer.

Example 3.1. **`OutputBuffer`** and **`InputBuffer`**

```
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;
```

```
};
```

The **`InputBuffer`** and **`OutputBuffer`** classes maintain an internal array into which instances of the standard Java types can be contiguously packed or unpacked, using the `pack` or `unpack` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form called `network byte order` to make them machine independent.

Example 3.2. **`OutputObjectState`** and **`InputObjectState`**

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

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

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

3.1.2. The object store

The object store provided with TxCore deliberately has a fairly restricted interface so that it can be implemented in a variety of ways. For example, object stores are implemented in shared memory, on the Unix file system (in several different forms), and as a remotely accessible store. More complete information about the object stores available in TxCore can be found in the Appendix.

Note

As with all TxCore classes, the default object stores are pure Java implementations. to access the shared memory and other more complex object store implementations, you need to use native methods.

All of the object stores hold and retrieve instances of the class **`InputObjectState`** or **`OutputObjectState`**. These instances are named by the Uid and Type of the object that they represent. States are read using the `read_committed` operation and written by the system using

the `write_uncommitted` operation. Under normal operation 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` operation is invoked. Normally all interaction with the object store is performed by TxCore system components as appropriate thus the existence of any shadow versions of objects in the store are hidden from the programmer.

Example 3.3. StateStatus

```
public StateStatus
{
    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;
}
```

Example 3.4. ObjectStore

```
public abstract class ObjectStore
{
    /* 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 must make certain state changes persistent, so it can recover in the event of a failure and either continue to commit, or rollback. When using **TXOJ**, TxCore will take care of this automatically. To guarantee `ACID` properties, these state changes must be flushed to the persistence store implementation before the transaction can proceed to commit. Otherwise, the application may assume that the transaction has committed when in fact the state changes may still reside within an operating system cache, and may be lost by a subsequent machine failure. By default, TxCore ensures that such state changes are flushed. However, doing so can impose a significant performance penalty on the application.

To prevent transactional object state flushes, set the `ObjectStoreEnvironmentBean.objectStoreSync` variable to **`OFF`**.

3.1.3. Selecting an object store implementation

TxCore comes with support for several different object store implementations. The Appendix describes these implementations, how to select and configure a given implementation on a per-object basis using the `ObjectStoreEnvironmentBean.objectStoreType` property variable, and indicates how additional implementations can be provided.

3.1.3.1. StateManager

The TxCore class **`StateManager`** manages the state of an object and provides all of the basic support mechanisms required by an object for state management purposes. **`StateManager`** is responsible for creating and registering appropriate resources concerned with the persistence

and recovery of the transactional object. If a transaction is nested, then **`StateManager`** will also propagate these resources between child transactions and their parents at commit time.

Objects are assumed to be of three possible flavors.

Three Flavors of Objects

Recoverable

`StateManager` attempts to generate and maintain appropriate recovery information for the object. Such objects have lifetimes that do not exceed the application program that creates them.

Recoverable and Persistent

The lifetime of the object is assumed to be greater than that of the creating or accessing application, so that 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 appropriate times.

Neither Recoverable nor Persistent

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

This object property is selected at object construction time and cannot be changed thereafter. Thus an object cannot gain (or lose) recovery capabilities at some arbitrary point during its lifetime.

Example 3.5. Object Store Implementation Using **`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 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, int objectModel);
    protected StateManager (Uid uid);
    protected StateManager (Uid uid, int objectModel);
    . . .
```

```
        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`** will invoke the operations `save_state` (while performing deactivation), `restore_state` (while performing activation), and `type` at various points during the execution of the application. These operations must be implemented by the programmer since **`StateManager`** does not have access to a runtime description of the layout of an arbitrary Java object in memory and thus cannot implement a default policy for converting the in memory version of the object to its passive form. However, the capabilities provided by **`InputObjectState`** and **`OutputObjectState`** make the writing of these routines fairly simple. For example, the `save_state` implementation for a class **`Example`** that had member variables called `A`, `B`, and `C` could simply be *Example 3.6, "Example Implementation of Methods for **`StateManager`**"*.

Example 3.6. Example Implementation of Methods for **`StateManager`**

```
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;
        }
}
```

In order 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 is used to determine the location in the object store where the state of instances of that class will be saved and ultimately restored. This location can actually be any valid string. However, you should avoid using the hash character (#) as this is reserved for special directories that TxCore requires.

The `get_uid` operation of **`StateManager`** provides read-only access to an object's internal system name for whatever purpose the programmer requires, such as registration of the name in a name server. The value of the internal system name can only be set when an object is initially constructed, either by the provision of an explicit parameter or by generating a new identifier when the object is created.

The `destroy` method can be used to remove the object's state from the object store. This is an atomic operation, and therefore will only remove the state if the top-level transaction within which it is invoked eventually commits. The programmer must obtain exclusive access to the object prior to invoking this operation.

Since object recovery and persistence essentially have complimentary requirements (the only difference being where state information is stored and for what purpose), **`StateManager`** effectively combines the management of these two properties into a single mechanism. It uses instances of the classes **`InputObjectState`** and **`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. This allows different information to be saved for recovery and persistence purposes.

3.1.3.2. Object models

TxCore supports two models for objects, which affect how an objects state and concurrency control are implemented.

TxCore Object Models

Single

Only a single copy of the object exists within the application. This copy resides within a single JVM, and all clients must address their invocations to this server. This model provides better performance, but represents a single point of failure, and in a multi-threaded environment may not protect the object from corruption if a single thread fails.

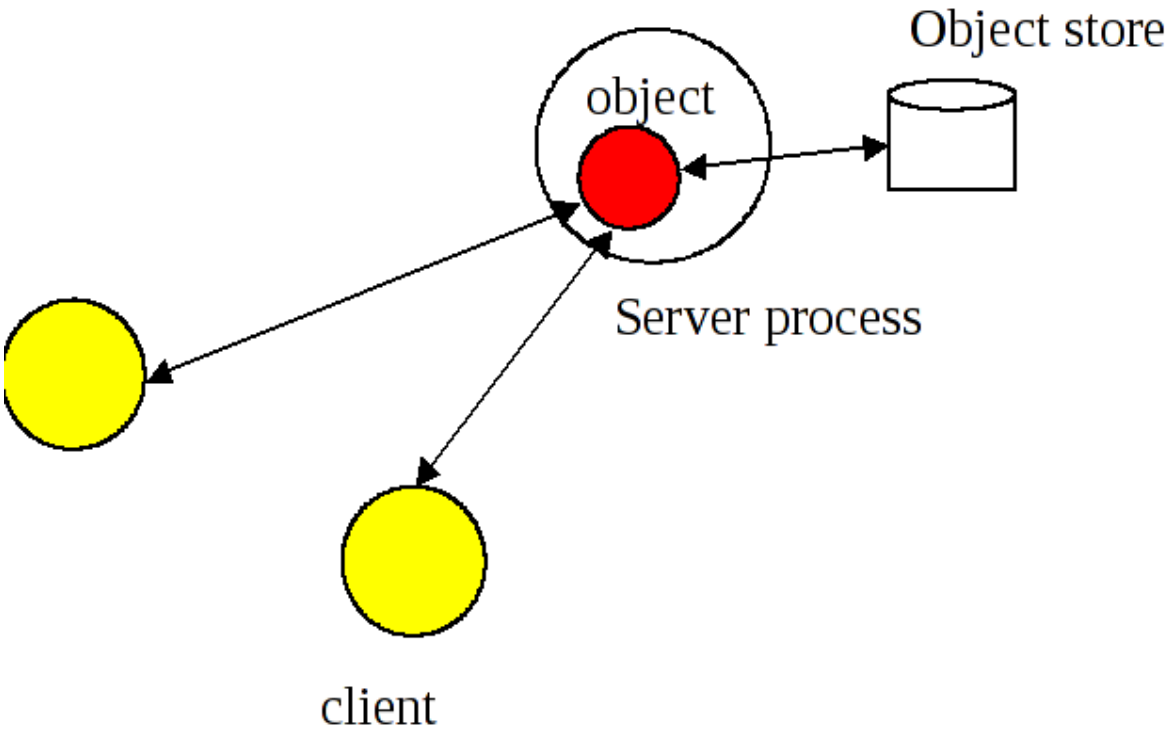


Figure 3.1. Single Object Model

Multiple
: Logically, a single instance of the object exists, but copies of it are distributed across different JVMs. The performance of this model is worse than the SINGLE model, but it provides better failure isolation.

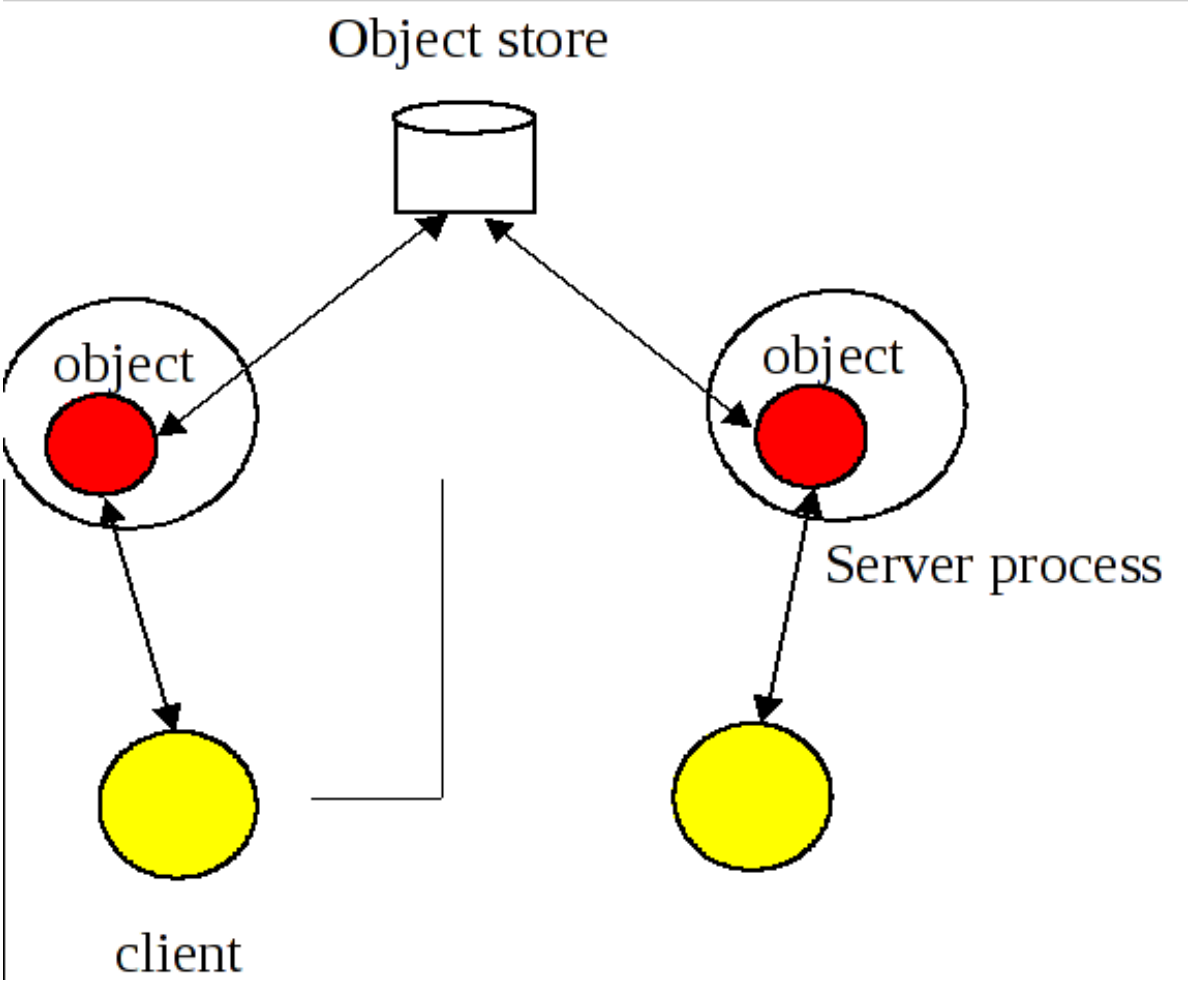


Figure 3.2. Multiple Object Model

The default model is SINGLE. The programmer can override this on a per-object basis by using the appropriate constructor.

3.1.3.3. Summary

In summary, the TxCore class **`StateManager`** manages the state of an object and provides all of the basic support mechanisms required by an object for state management purposes. Some operations must be defined by the class developer. These operations are: `save_state`, `restore_state`, and `type`.

boolean `save_state`(OutputObjectState `state`, int`ObjectType`)
: Invoked whenever the state of an object might need to be saved for future use, primarily for recovery or persistence purposes. The *`ObjectType`* parameter indicates the reason that `save_state` was invoked by TxCore. This enables the programmer to save different pieces of information into the **`OutputObjectState`** supplied as the first parameter depending upon whether the state is needed for recovery or persistence purposes. For example, pointers to other TxCore objects might be saved simply as pointers for recovery purposes but as Uids for persistence purposes. As shown earlier, the **`OutputObjectState`** class provides convenient operations to allow the saving of instances of all of the basic types in Java. In order to support

crash recovery for persistent objects it is necessary for all `save_state` methods to call `super.save_state`.

`save_state` assumes that an object is internally consistent and that all variables saved have valid values. It is the programmer's responsibility to ensure that this is the case.

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

Invoked whenever the state of an object needs to be restored to the one supplied. Once again the second parameter allows different interpretations of the supplied state. In order to support crash recovery for persistent objects it is necessary for all `restore_state` methods to call `super.restore_state`.

String `type` ()

The TxCore persistence mechanism requires a means of determining the type of an object as a string so that it can save or restore the state of the object into or from the object store. By convention this information indicates the position of the class in the hierarchy. For example, **`/StateManager/LockManager/Object`**.

The `type` method is used to determine the location in the object store where the state of instances of that class will be saved and ultimately restored. This can actually be any valid string. However, you should avoid using the hash character (#) as this is reserved for special directories that TxCore requires.

3.1.3.4. Example

Consider the following basic **`Array`** class derived from the **`StateManager`** class. In this example, to illustrate saving and restoring of an object's state, the `highestIndex` variable is used to keep track of the highest element of the array that has a non-zero value.

Example 3.7. **`Array`** Class

```
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";
}
```

3.2. Lock management and concurrency control

Concurrency control information within TxCore is maintained by locks. Locks which are required to be shared between objects in different processes may be held within a lock store, similar to the object store facility presented previously. The lock store provided with TxCore deliberately has a fairly restricted interface so that it can be implemented in a variety of ways. For example, lock stores are implemented in shared memory, on the Unix file system (in several different forms), and as a remotely accessible store. More information about the object stores available in TxCore can be found in the Appendix.

Note

As with all TxCore classes, the default lock stores are pure Java implementations. To access the shared memory and other more complex lock store implementations it is necessary to use native methods.

Example 3.8. **`LockStore`**

```
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);
};
```

3.2.1. Selecting a lock store implementation

TxCore comes with support for several different object store implementations. If the object model being used is SINGLE, then no lock store is required for maintaining locks, since the information about the object is not exported from it. However, if the MULTIPLE model is used, then different run-time environments (processes, Java virtual machines) may need to share concurrency control information. The implementation type of the lock store to use can be specified for all objects within a given execution environment using the `TxojEnvironmentBean.lockStoreType` property variable. Currently this can have one of the following values:

BasicLockStore
: This is an in-memory implementation which does not, by default, allow sharing of stored information between execution environments. The application programmer is responsible for sharing the store information.

BasicPersistentLockStore
: This is the default implementation, and stores locking information within the local file system. Therefore 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 TxCore installation directory. You can override this at runtime by setting the `TxojEnvironmentBean.lockStoreDir` property variable accordingly, or placing the location within the `CLASSPATH`.

```
java -D TxojEnvironmentBean.lockStoreDir=/var/tmp/LockStore myprogram
```

```
java –classpath $CLASSPATH;/var/tmp/LockStore myprogram
```

If neither of these approaches is taken, then the default location will be at the same level as the **`etc`** directory of the installation.

3.2.2. LockManager

The concurrency controller is implemented by the class **`LockManager`**, which provides sensible default behavior, while allowing the programmer to override it if deemed necessary by the particular semantics of the class being programmed. The primary programmer interface to the concurrency controller is via the `setlock` operation. By default, the TxCore runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. Lock acquisition is under programmer control, since just as **`StateManager`** cannot determine if an operation modifies an object, **`LockManager`** cannot determine if an operation requires a read or write lock. Lock release, however, is normally under control of the system and requires no further intervention by the programmer. This ensures that the two-phase property can be correctly maintained.

The **`LockManager`** class is primarily responsible for managing requests to set a lock on an object or to release a lock as appropriate. However, since it is derived from **`StateManager`**, it can also control when some of the inherited facilities are invoked. For example, if a request to set a write lock is granted, then **`LockManager`** invokes modified directly assuming that the setting of a write lock implies that the invoking operation must be about to modify the object. This may in turn cause recovery information to be saved if the object is recoverable. In a similar fashion, successful lock acquisition causes activate to be invoked.

Therefore, **`LockManager`** is directly responsible for activating and deactivating persistent objects, as well as registering `Resources` for managing concurrency control. By driving the **`StateManager`** class, it is also responsible for registering `Resources` for persistent or recoverable state manipulation and object recovery. The application programmer simply sets appropriate locks, starts and ends transactions, and extends the `save_state` and `restore_state` methods of **`StateManager`**.

Example 3.9. **`LockResult`**

```
 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, int objectModel);
```

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

The `setlock` operation must be parametrized with the type of lock required (READ or WRITE), and the number of retries to acquire the lock before giving up. If a lock conflict occurs, one of the following scenarios will take place:

- If the retry value is equal to `LockManager.waitTotalTimeout`, then the thread which called `setlock` will be blocked until the lock is released, or the total timeout specified has elapsed, and in which **REFUSED** will be returned.
- If the lock cannot be obtained initially then **`LockManager`** will try for the specified number of retries, 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. The time between retries is specified in micro-seconds.
- If a lock conflict occurs the current implementation simply times out lock requests, thereby preventing deadlocks, rather than providing a full deadlock detection scheme. If the requested lock is obtained, the `setlock` operation will return the value GRANTED, otherwise the value **REFUSED** is returned. It is the responsibility of the programmer to ensure that the remainder of the code for an operation is only executed if a lock request is granted. Below are examples of the use of the setlock operation.

Example 3.10. `setlock` Method Usage

```
 res = setlock(new Lock(WRITE), 10); // Will attempt to set a
                  // write lock 11 times (10
                  // retries) on the object
                  // before giving up.
res = setlock(new Lock(READ), 0);      // Will attempt to set a read
                  // lock 1 time (no retries) on
                  // the object before giving up.
res = setlock(new Lock(WRITE);      // Will attempt to set a write
                  // lock 101 times (default of
                  // 100 retries) on the object
                  // before giving up.
```

The concurrency control mechanism is integrated into the atomic action mechanism, thus ensuring that as locks are granted on an object appropriate information is registered with the currently running atomic action to ensure that the locks are released at the correct time. This frees the programmer from the burden of explicitly freeing any acquired locks if they were acquired within atomic actions. However, if locks are acquired on an object outside of the scope of an atomic action, it is the programmer's responsibility to release the locks when required, using the corresponding `releaselock` operation.

3.2.3. Locking policy

Unlike many other systems, locks in TxCore are not special system types. Instead they are simply instances of other TxCore objects (the class **`Lock`** which is also derived from **`StateManager`** so that locks may be made persistent if required and can also be named in a simple fashion). Furthermore, **`LockManager`** deliberately has no knowledge of the semantics of the actual policy by which lock requests are granted. Such information is maintained by the actual **`Lock`** class instances which provide operations (the `conflictsWith` operation) by which **`LockManager`** can determine if two

locks conflict or not. This separation is important in that it allows the programmer to derive new lock types from the basic **Lock** class and by providing appropriate definitions of the conflict operations enhanced levels of concurrency may be possible.

Example 3.11. **LockMode** Class

```
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` operation which **LockManager** uses to determine if granting this locking request requires a call on modified. This operation is provided so that locking modes other than simple read and write can be supported. The supplied **Lock** class supports the traditional multiple reader/single writer policy.

3.2.4. Object constructor and destructor

Recall that TxCore objects can be recoverable, recoverable and persistent, or neither. Additionally each object possesses a unique internal name. These attributes can only be set when that object is constructed. Thus **LockManager** provides two protected constructors for use by derived classes, each of which fulfills a distinct purpose

Protected Constructors Provided by **LockManager**

`LockManager ()`

This constructor allows the creation of new objects, having no prior state.

`LockManager` (int `ObjectType`, int `objectModel`)

As above, this constructor allows the creation of new objects having no prior state. exist. The `ObjectType` parameter determines whether an object is simply recoverable (indicated by **RECOVERABLE**), recoverable and persistent (indicated by **ANDPERSISTENT**), or neither (indicated by **NEITHER**). If an object is marked as being persistent then the state of the object will be stored in one of the object stores. The shared parameter only has meaning if it is **RECOVERABLE**. If the object model is **SINGLE** (the default behavior) then the recoverable state of the object is maintained within the object itself, and has no external representation). Otherwise an in-memory (volatile) object store is used to store the state of the object 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. Later examples in this chapter illustrate this point further.

`LockManager`(Uid `objUid`)

This constructor allows access to an existing persistent object, whose internal name is given by the `objUid` parameter. Objects constructed using this operation will normally have their prior state (identified by `objUid`) loaded from an object store automatically by the system.

`LockManager`(Uid `objUid`, int `objectModel`)

As above, this constructor allows access to an existing persistent object, whose internal name is given by the `objUid` parameter. Objects constructed using this operation will normally have their prior state (identified by `objUid`) loaded from an object store automatically by the system. If the object model is **`SINGLE`** (the default behavior), then the object will not be reactivated at the start of each top-level transaction.

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

Advanced transaction issues with TxCore

Atomic actions (transactions) can be used by both application programmers and class developers. Thus entire operations (or parts of operations) can be made atomic as required by the semantics of a particular operation. This chapter will describe some of the more subtle issues involved with using transactions in general and TxCore in particular.

4.1. Last resource commit optimization (LRCO)

In some cases it may be necessary to enlist participants that are not two-phase commit aware into a two-phase commit transaction. If there is only a single resource then there is no need for two-phase commit. However, if there are multiple resources in the transaction, the *Last Resource Commit optimization (LRCO)* comes into play. It is possible for a single resource that is one-phase aware (i.e., can only commit or roll back, with no prepare), to be enlisted in a transaction with two-phase commit aware resources. The coordinator treats the one-phase aware resource slightly differently, in that it executes the prepare phase on all other resource first, and if it then intends to commit the transaction it passes control to the one-phase aware resource. If it commits, then the coordinator logs the decision to commit and attempts to commit the other resources as well.

In order to utilize the LRCO, your participant must implement the `com.arjuna.ats.arjuna.coordinator.OnePhase` interface and be registered with the transaction through the `BasicAction.add` operation. Since this operation expects instances of **`AbstractRecord`**, you must create an instance of **`com.arjuna.ats.arjuna.LastResourceRecord`** and give your participant as the constructor parameter.

Example 4.1. Class **`com.arjuna.ats.arjuna.LastResourceRecord`**

```
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();
    }
```

4.2. Nested transactions

There are no special constructs for nesting of transactions. If an action is begun while another action is running then it is automatically nested. This allows for a modular structure to applications, whereby objects can be implemented using atomic actions within their operations without the application programmer having to worry about the applications which use them, and whether or not the applications will use atomic actions as well. Thus, in some applications actions may be top-level, whereas in others they may be nested. Objects written in this way can then be shared between application programmers, and TxCore will guarantee their consistency.

If a nested action is aborted, all of its work will be undone, although strict two-phase locking means that any locks it may have obtained will be retained until the top-level action commits or aborts. If a nested action commits then the work it has performed will only be committed by the system if the top-level action commits. If the top-level action aborts then all of the work will be undone.

The committing or aborting of a nested action does not automatically affect the outcome of the action within which it is nested. This is application dependent, and allows a programmer to structure atomic actions to contain faults, undo work, etc.

4.3. Asynchronously committing a transaction

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

- In the case of many registered resources, the `prepare` operating can 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`, possibly many prepare operations will have been made needlessly.
- In the case where heuristic reporting is not required by the application, the second phase of the commit protocol can be done asynchronously, since its success or failure is not important.

Therefore, JBoss Transaction Service provides runtime options to enable possible threading optimizations. By setting the `CoordinatorEnvironmentBean.asyncPrepare` environment variable to **YES**, during the `prepare` phase a separate thread will be created for each registered participant within the transaction. By setting `CoordinatorEnvironmentBean.asyncCommit` to **YES**, a separate thread will be created to complete the second phase of the transaction if knowledge about heuristics outcomes is not required.

4.4. Independent top-level transactions

In addition to normal top-level and nested atomic actions, TxCore also supports independent top-level actions, which can be used to relax strict serializability in a controlled manner. An independent top-level action can be executed from anywhere within another atomic action and behaves exactly like a normal top-level action. Its results are made permanent when it commits and will not be undone if any of the actions within which it was originally nested abort.

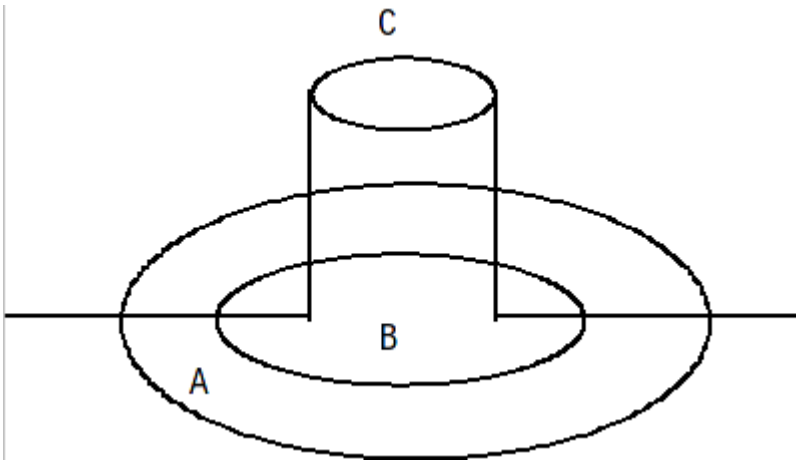


Figure 4.1. Independent Top-Level Action

Top-level actions can be used within an application by declaring and using instances of the class **`TopLevelTransaction`**. They are used in exactly the same way as other transactions.

4.5. Transactions within save_state and restore_state methods

Exercise caution when writing the save_state and restore_state operations to ensure that no atomic actions are started, either explicitly in the operation or implicitly through use of some other operation. This restriction arises due to the fact that TxCore may invoke restore_state as part of its commit processing resulting in the attempt to execute an atomic action during the commit or abort phase of another action. This might violate the atomicity properties of the action being committed or aborted and is thus discouraged.

Example 4.2.

If we consider the *Example 3.7, "**Array** Class"* given previously, the set and get operations could be implemented as shown below.

This is a simplification of the code, ignoring error conditions and exceptions.

```
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;
}
```

4.6. Garbage collecting objects

Java objects are deleted when the garbage collector determines that they are no longer required. Deleting an object that is currently under the control of a transaction must be approached with caution since if the object is being manipulated within a transaction its fate is effectively determined by the transaction. Therefore, regardless of the references to a transactional object maintained by an application, TxCore will always retain its own references to ensure that the object is not garbage collected until after any transaction has terminated.

4.7. Transaction timeouts

By default, transactions live until they are terminated by the application that created them or a failure occurs. However, it is possible to set a timeout (in seconds) on a per-transaction basis such that if the transaction has not terminated before the timeout expires it will be automatically rolled back.

In TxCore, the timeout value is provided as a parameter to the `AtomicAction` constructor. If a value of **`AtomicAction.NO_TIMEOUT`** is provided (the default) then the transaction will not be automatically timed out. Any other positive value is assumed to be the timeout for the transaction (in seconds). A value of zero is taken to be a global default timeout, which can be provided by the property `CoordinatorEnvironmentBean.defaultTimeout`, which has a default value of 60 seconds.

Note

Default timeout values for other JBoss Transaction Service components, such as JTS, may be different and you should consult the relevant documentation to be sure.

When a top-level transaction is created with a non-zero timeout, it is subject to being rolled back if it has not completed within the specified number of seconds. JBoss Transaction Service uses a separate reaper thread which monitors all locally created transactions, and forces them to roll back if their timeouts elapse. If the transaction cannot be rolled back at that point, the reaper will force it into a rollback-only state so that it will eventually be rolled back.

By default this thread is dynamically scheduled to awake according to the timeout values for any transactions created, ensuring the most timely termination of transactions. It may alternatively be configured to awake at a fixed interval, which can reduce overhead at the cost of less accurate rollback timing. For periodic operation, change the `CoordinatorEnvironmentBean.txReaperMode` property from its default value of **`DYNAMIC`** to **`PERIODIC`** and set the interval between runs, in milliseconds, using the property `CoordinatorEnvironmentBean.txReaperTimeout`. The default interval in **`PERIODIC`** mode is 120000 milliseconds.

Warning

In earlier versions the **PERIODIC** mode was known as **NORMAL** and was the default behavior. The use of the configuration value **NORMAL** is deprecated and **PERIODIC** should be used instead if the old scheduling behavior is still required.

If a value of **0** is specified for the timeout of a top-level transaction, or no timeout is specified, then JBoss Transaction Service will not impose any timeout on the transaction, and the transaction will be allowed to run indefinitely. This default timeout can be overridden by setting the `CoordinatorEnvironmentBean.defaultTimeout` property variable when using to the required timeout value in seconds, when using ArjunaCore, ArjunaJTA or ArjunaJTS.

Note

As of JBoss Transaction Service 4.5, transaction timeouts have been unified across all transaction components and are controlled by ArjunaCore.

4.7.1. Monitoring transaction timeouts

If you want to be informed when a transaction is rolled back or forced into a rollback-only mode by the reaper, you can create a class that inherits from class **`com.arjuna.ats.arjuna.coordinator.listener.ReaperMonitor`** and overrides the `rolledBack` and `markedRollbackOnly` methods. When registered with the reaper via the `TransactionReaper.addListener` method, the reaper will invoke one of these methods depending upon how it tries to terminate the transaction.

Note

The reaper will not inform you if the transaction is terminated (committed or rolled back) outside of its control, such as by the application.

Hints and tips

5.1. General

5.1.1. Using transactions in constructors

Examples throughout this manual use transactions in the implementation of constructors for new persistent objects. This is deliberate because 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. If, however, the constructor transaction commits but is nested because another transaction that was started prior to object creation is running, the state is written only if all of the parent transactions commit.

On the other hand, if the constructor does not use transactions, inconsistencies in the system can arise. 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 some transaction.

Example 5.1. Nested Transactions In Constructors

```
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
```

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` operation of `obj2` is invoked, the object will be activated and the Uid of `obj1` remembered. 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 an action later in the application, it will never be saved. If, however, the constructor had used an atomic action, the state of `obj1` would have automatically been saved at the time it was constructed and this inconsistency could not arise.

5.1.2. `save_state` and `restore_state` methods

TxCore may invoke the user-defined `save_state` operation 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 particularly a possibility if it uses atomic actions. It is important, therefore, that all of the variables saved by `save_state` are correctly initialized. Exercise caution when writing the `save_state` and `restore_state` operations, to ensure that no transactions are started, either explicitly in the operation, or implicitly through use of some other operation. The reason for this restriction is that TxCore may invoke `restore_state` as part of its commit processing. This would result in the attempt to execute an atomic transaction during the commit or abort phase of another transaction. This might violate the atomicity properties of the transaction being committed or aborted, and is thus discouraged. In order 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`.

5.1.2.1. Packing objects

All of the basic types of Java (int, long, etc.) can be saved and restored from an **`InputObjectState`** or **`OutputObjectState`** instance by using the `pack` and `unpack` routines provided by **`InputObjectState`** and **`OutputObjectState`**. However packing and unpacking objects should be handled differently. This is because packing objects brings in the additional problems of aliasing. Aliasing happens when two different object references may point at the same item. For example:

Example 5.2. Aliasing

```
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 at the same string. A naive implementation of `save_state` might copy the string twice. From a `save_state` perspective this is simply inefficient. However, `restore_state` would unpack the two strings into different areas of memory, destroying the original aliasing information. The current version of TxCore packs and unpacks separate object references.

5.2. Direct use of StateManager

The examples throughout this manual derive user classes from **`LockManager`**. These are two important reasons for this.

1. Firstly, and most importantly, the serializability constraints of atomic actions require it.
2. It reduces the need for programmer intervention.

However, if you only require access to TxCore's persistence and recovery mechanisms, direct derivation of a user class from **`StateManager`** is possible.

Classes derived directly from **`StateManager`** must make use of its state management mechanisms explicitly. These interactions are normally undertaken by **`LockManager`**. From a programmer's point of view this amounts to making appropriate use of the operations `activate`, `deactivate`, and `modified`, since **`StateManager`**'s constructors are effectively identical to those of **`LockManager`**.

Example 5.3. `activate`

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

Activate loads an object from the object store. The object's UID must already have been set via the constructor and the object must exist in the store. If the object is successfully read then

restore_state is called to build the object in memory. Activate is idempotent so 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. A value of null means use the default store.

Example 5.4. `deactivate`

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

The inverse of activate. First calls save_state to build the compacted image of the object which is then saved in the object store. Objects are only saved if they have been modified since they were activated. The parameter represents the root name of the object store into which the object should be saved. A value of null means use the default store.

Example 5.5. `modified`

```
void modified ()
```

Must be called prior to modifying the object in memory. If it is not called, the object will not be saved in the object store by `deactivate`.

Constructing a Transactional Objects for Java application

Development Phases of a TxCore Application

1. First, develop new classes with characteristics like persistence, recoverability, and concurrency control.

2. Then develop the applications that make use of the new classes of objects.

Although these two phases may be performed in parallel and by a single person, this guide refers to the first step as the job of the class developer, and the second as the job of the applications developer. The class developer defines appropriate `save_state` and `restore_state` operations for the class, sets appropriate locks in operations, and invokes the appropriate TxCore class constructors. The applications developer defines the general structure of the application, particularly with regard to the use of atomic actions.

This chapter outlines a simple application, a simple FIFO Queue class for integer values. The Queue is implemented with a doubly linked list structure, and is implemented as a single object. This example is used throughout the rest of this manual to illustrate the various mechanisms provided by TxCore. Although this is an unrealistic example application, it illustrates all of the TxCore modifications without requiring in depth knowledge of the application code.

Note

The application is assumed not to be distributed. To allow for distribution, context information must be propagated either implicitly or explicitly.

6.1. 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 on to the queue (`enqueue`) and to be removed from it (`dequeue`), and values of elements in the queue can also be changed or inspected. In this example implementation, an array represents the queue. A limit of `QUEUE_SIZE` elements has been imposed for this example.

Example 6.1. Java interface definition of class **queue**

```
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;
};
```

6.2. Constructors and destructors

Using an existing persistent object requires the use of a special constructor that takes the Uid of the persistent object, as shown in *Example 6.2, "Class **`TransactionalQueue`**"*.

Example 6.2. Class **`TransactionalQueue`**

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

    numberOfElements = 0;
}
The constructor that creates a new persistent object is similar:
    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);
        }
}
```

The use of an atomic action within the constructor for a new object follows the guidelines outlined earlier and ensures that the object's state will be written to the object store when the appropriate top level atomic action commits (which will either be the action A or some enclosing action active when the TransactionalQueue was constructed). The use of atomic actions in a constructor is simple: an action must first be declared and its begin operation invoked; the operation must then set an appropriate lock

on the object (in this case a WRITE lock must be acquired), then the main body of the constructor is executed. If this is successful the atomic action can be committed, otherwise it is aborted.

The destructor of the **queue** class is only required to call the `terminate` operation of **`LockManager`**.

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

6.3. Required methods

6.3.1. `save_state`, `restore_state`, and `type`

Example 6.3. Method `save_state`

```
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 6.4. Method `restore_state`

```
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 6.5. Method `type`

Because the Queue class is derived from the LockManager class, the operation type should be:

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

6.3.2. enqueue and dequeue methods

If the operations of the **queue** class are to be coded as atomic actions, then the enqueue operation might have the structure given below. The `dequeue` operation is similarly structured, but is not implemented here.

Example 6.6. Method `enqueue`

```
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();
        }
}
```

6.3.3. queueSize method

Example 6.7. Method queueSize

```
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;
}
```

6.3.4. inspectValue and setValue methods

Note

The setValue method is not implemented here, but is similar in structure to *Example 6.8, "Method inspectValue"*.

Example 6.8. Method inspectValue

```
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;
}
```

6.4. The client

Rather than show all of the code for the client, this example concentrates on a representative portion. Before invoking operations on the object, the client must first bind to the object. In the local case this simply requires the client to create an instance of the object.

Example 6.9. Binding to the Object

```
 public static void main (String[] args)
{
    TransactionalQueue myQueue = new TransactionalQueue();
    Before invoking one of the queue's operations, the client starts a transaction. The
 queueSize operation is shown below:
    AtomicAction A = new AtomicAction();
    int size = 0;

    try
        {
            A.begin(0);

            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!");
        }
}
```

6.5. Comments

Since the queue object is persistent, the state of the object survives any failures of the node on which it is located. The state of the object that survives is the state produced by the last top-level committed atomic action performed on the object. If an application intends to perform two enqueue operations atomically, for example, you can nest the enqueue operations in another enclosing atomic action. In addition, concurrent operations on such a persistent object are serialized, preventing inconsistencies in the state of the object.

However, since the elements of the queue objects are not individually concurrency controlled, certain combinations of concurrent operation invocations are executed serially, even though logically they could be executed concurrently. An example of this is modifying the states of two different elements in the queue. The platform Development Guide addresses some of these issues.

Appendix A. Object store implementations

A.1. The ObjectStore

This appendix examines the various TxCore object store implementations and gives guidelines for creating other implementations and plugging into an application.

This release of JBoss Transaction Service contains several different implementations of a basic object store. Each serves a particular purpose and is generally optimized for that purpose. Each of the implementations is derived from the `ObjectStore` interface, which defines the minimum operations which must be provided for an object store implementation to be used by the Transaction Service. You can override the default object store implementation at runtime by setting the `com.arjuna.ats.arjuna.objectstore.objectStoreType` property variable to one of the types described below.

Example A.1. Class **`StateStatus`**

```
/*
 * 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 StateStatus
{
    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 class StateType
{
    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 abstract class ObjectStore implements BaseStore, ParticipantStore,
                                              RecoveryStore, TxLog
{
    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);
};
```

JBoss Transaction Service programmers do not usually need to interact with any of the object store implementations directly, apart from possibly creating them in the first place. Even this is not necessary if the default store type is used, since JBoss Transaction Service creates stores as necessary. All stores manipulate instances of the class **`ObjectState`**. These instances are named using a `type` (via the object's `type()` operation) and a `Uid`.

For atomic actions purposes, object states in the store can be principally in two distinct states: **`OS_COMMITTED`** or **`OS_UNCOMMITTED`**. An object state starts in the **`OS_COMMITTED`** state, but when it is 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 handle these state transitions by making use of shadow copies of object states. However, any other implementation that maintains this abstraction is permissible.

Object states may become hidden, and thus inaccessible, under the control of the crash recovery system.

You can browse the contents of a store through the `allTypes` and `allObjUids` operations. `allTypes` returns an **`InputObjectState`** containing all of the type names of all objects in a store, terminated by a null name. `allObjUids` returns an **`InputObjectState`** containing all of the Uids of all objects of a given type, terminated by the special `Uid.nullUid()`.

A.1.1. 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.

A.1.1.1. 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, as given by the `type()` operation, determines the directory into which the object is placed.
- All of the stores have a common root directory that is determined when JBoss Transaction Service is configured. This directory name is automatically prepended to any store-specific root information.
- All stores also have the notion of 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 name is **`defaultStore`**.

```
<ObjectStore root Directory from configure>     <filename>/JBossTS/ObjectStore/</filename>
   <ObjectStore Type1>                          <filename>FragmentedStore/</filename>
      <Default root>                            <filename>defaultStore/</filename>
         <StateManager>                         <filename>StateManager</filename>
            <LockManager>                       <filename>LockManager/</filename>
               <User Types>
      <Localised root 2>                        <filename>myStore/</filename>
         <StateManager>                         <filename>StateManager/</filename>

   <ObjectStore Type2>                          <filename>ActionStore/</filename>
         <Default root>                         <filename>defaultStore/</filename>
```

A.1.1.2. The shadowing store

The shadowing store s the original version of the object store, which was provided in prior releases. It is implemented by the class **`ShadowingStore`**. It is simple but slow. It uses pairs of files to represent objects. One file is the shadow version and the other is the committed version. Files are opened, locked, operated upon, unlocked, and closed on every interaction with the object store. This causes a lot of I/O overhead.

If you are overriding the object store implementation, the type of this object store is **`ShadowingStore`**.

A.1.1.3. No file-level locking

Since transactional objects are concurrency-controlled through **`LockManager`**, you do not need to impose additional locking at the file level. The basic ShadowingStore implementation handles file-level locking. Therefore, the default object store implementation for JBoss Transaction Service, `ShadowNoFileLockStore`, relies upon user-level locking. This enables it to provide better performance than the `ShadowingStore` implementation.

If you are overriding the object store implementation, the type of this object store is **`ShadowNoFileLockStore`**.

A.1.1.4. The hashed store

The HashedStore has the same structure for object states as the ShadowingStore, but has an alternate directory structure that is better suited to storing large numbers of objects of the same type. Using this store, objects are scattered among a set of directories by applying a hashing function to the object's Uid. By default, 255 sub-directories are used. However, you can override this by setting the `ObjectStoreEnvironmentBean.hashedDirectories` environment variable accordingly.

If you are overriding the object store implementation, the type of this object store is **`HashedStore`**.

A.1.1.5. The JDBC store

The JDBCStore uses a JDBC database to save persistent object states. When used in conjunction with the Transactional Objects for Java API, nested transaction support is available. In the current

implementation, all object states are stored as *Binary Large Objects (BLOBs)* within the same table. The limitation on object state size imposed by using BLOBs is **64k**. If you try to store an object state which exceeds this limit, an error is generated and the state is not stored. The transaction is subsequently forced to roll back.

When using the JDBC object store, the application must provide an implementation of the `JDBCAccess` interface, located in the com.arjuna.ats.arjuna.objectstore package:

Example A.2. Interface `JDBCAccess`

```
 public interface JDBCAccess
{
    public Connection getConnection () throws SQLException;
    public void putConnection (Connection conn) throws SQLException;
    public void initialise (Object[] objName);
}
```

The implementation of this class is responsible for providing the **`Connection`** which the JDBC ObjectStore uses to save and restore object states:

`getConnection`
: Returns the Connection to use. This method is 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
: Used to pass additional arbitrary information to the implementation.

The JDBC object store initially requests the number of **`Connections`** defined in the `ObjectStoreEnvironmentBean.jdbcPoolSizeInitial` property and will use no more than defined in the `ObjectStoreEnvironmentBean.jdbcPoolSizeMaximum property`.

The implementation of the `JDBCAccess` interface to use should be set in the `ObjectStoreEnvironmentBean.jdbcUserDbAccessClassName` property variable.

If overriding the object store implementation, the type of this object store is JDBCStore.

A JDBC object store can be used for managing the transaction log. In this case, the transaction log implementation should be set to **`JDBCActionStore`** and the **`JDBCAccess`** implementation must be provided via the `ObjectStoreEnvironmentBean.jdbcTxDbAccessClassName property` variable. In this case, the default table name is `JBossTSTxTable`.

You can use the same **`JDBCAccess`** implementation for both the user object store and the transaction log.

A.1.1.6. The cached store

This object 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 and either flushes the cache periodically or when it is full. The failure semantics associated with this object store are different from the normal persistent object stores, because a failure could result in states in the cache being lost.

If overriding the object store implementation, the type of this object store is CacheStore.

Configuration Properties

ObjectStoreEnvironmentBean.cacheStoreHash
: sets the number of internal stores to hash the states over. The default value is 128.

ObjectStoreEnvironmentBean.cacheStoreSize
: the maximum size the cache can reach before a flush is triggered. The default is 10240 bytes.

ObjectStoreEnvironmentBean.cacheStoreRemovedItems
: 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 will simply remove the state from the cache, but leave a blank entry (rather than remove the entry immediately, which would affect the performance of the cache). When triggered, these entries are removed from the cache. The default value is twice the size of the hash.

ObjectStoreEnvironmentBean.cacheStoreWorkItems
: the maximum number of items that are allowed to build up in the cache before it is flushed. The default value is 100. `ObjectStoreEnvironmentBean.cacheStoreScanPeriod` sets the time in milliseconds for periodically flushing the cache. The default is 120 seconds.

ObjectStoreEnvironmentBean.cacheStoreSync
: determines whether flushes of the cache are sync-ed to disk. The default is **`OFF`**. To enable, set to **`ON`**.

A.1.1.7. LogStore

This implementation is based on a traditional transaction log. All transaction states within the same process (VM instance) are written to the same log (file), which is an append-only entity. When transaction data would normally be deleted, at the end of the transaction, a `delete` record is added to the log instead. Therefore, the log just keeps growing. Periodically a thread runs to prune the log of entries that have been deleted.

A log is initially given a maximum capacity beyond which it cannot grow. After it reaches this size, the system creates a new log for transactions that could not be accommodated in the original log. The new log and the old log are pruned as usual. During the normal execution of the transaction system, there may be an arbitrary number of log instances. These should be garbage collected by the system, (or the recovery sub-system, eventually.

Check the Configuration Options table for how to configure the LogStore.

Appendix B. Class definitions

This appendix contains an overview of those classes that the application programmer will typically use. The aim of this appendix is to provide a quick reference guide to these classes for use when writing applications in TxCore. For clarity only the public and protected interfaces of the classes will be given.

Example B.1. Class **LockManager**

```
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, int objectModel);
    protected LockManager (Uid storeUid);
    protected LockManager (Uid storeUid, int ot);
    protected LockManager (Uid storeUid, int ot, int objectModel);

    protected void terminate ();
};
```

Example B.2. Class **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, int objectModel);
    protected StateManager (Uid objUid);
    protected StateManager (Uid objUid, int ot);
    protected StateManager (Uid objUid, int ot, int objectModel);
    protected synchronized final void modified ();
};
```

Example B.3. Classes **`OutputObjectState`** and **`InputObjectState`**

```
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. Classes **`OutputBuffer`** and **`InputBuffer`**

```
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. Class **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. Class **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;
};
```

Appendix C. Revision History

Revision 0 **Fri Sep 24 2010** **Misty Stanley-Jones** *misty@redhat.com*

Convert existing documentation to Publican.
Update to 4.13.

Revision 0 **Thu Apr 14 2011** **Tom Jenkinson** *tom.jenkinson@redhat.com*

Moved some content to main developer's guide