The Architecture, Design and Implementation of ArjunaCore

Version No. 0.2

Revision Date: 26/03/2002

CONTENTS Page

1 Introduction 4
1.1 Scope 4
1.2 History 4
1.3 Terminology 4
1.4 References 4
2 Design goals 5
3 Failure Assumptions and Computation Model 6
3.1 Failure assumptions 6
3.2 Objects and transactions 6
3.2.1 Commit protocol 6
3.3 System Architecture 7
4 Implementing the System 8
4.1 The Life Cycle of an ArjunaCore Object 8
4.2 Implementing Object Storage Services 9
4.2.1 Saving object states 9
4.2.2 The Object Store 11
4.3 Implementing Atomic Transaction Services 11
4.3.1 Overview 11
4.3.2 Recovery and Persistence 13
4.3.3 The Concurrency Controller 15
4.3.4 Locking Policy 15
4.3.5 Co-ordinating Recovery, Persistence and Concurrency Control 16
4.4 Crash Recovery 19

1 Introduction

1.1 Scope

This document is a description of the Architecture, Design and Implementation of ArjunaCore 2.0, 2.1, 2.2 and 3.0.

The approved document forms a contract between the parties on the approvers' list and defines the expectations of all parties on the reviewers' list.

1.2 History

Date	Ver No.	Description	Updated By
18/3/2002	0.1	Draft	Mark Little
26/3/2002	0.2	Corrected document format and title	Steve Caughey

1.3 Terminology

Term	Description

1.4 References

References	Description

2 Design goals

ArjunaCore is an object-oriented programming system, implemented in 100% pure Java, that provides a set of tools for the construction of fault-tolerant applications using objects and transactions. *ArjunaCore* supports the computational model of *nested atomic transactions* controlling operations on persistent (long-lived) objects. Importantly, *ArjunaCore* is concerned solely with the use of local transactions, i.e., transactions that run on a single machine. If distributed transactions are required, *ArjunaCore* provides the necessary hooks to enable information about its local transactions (the *transaction context*) to be transmitted in a manner suitable for the environment in which it is running, e.g., CORBA IIOP or SOAP/XML.

At the heart of every transaction processing system is a *transaction manager* (also known as a *transaction coordinator*). It is the transaction manager that is responsible for ensuring the atomicity and durability properties of the transactions under its control. The isolation and consistency are provided by transactional resources that participate in the transaction on behalf of applications and services. The coordinator must maintain a transaction log in case of failures and a recovery system to use this log to complete transactions that were in flight and caught by any failures (e.g., a machine or process crash). It is important to realize that this functionality if required by *all* transaction systems, whether or not they support distributed transactions. *ArjunaCore* provides this exact functionality in a highly optimized, configurable and extensible manner. It has an extremely small footprint (easily executable on mobile devices, for example) and deliberately knows nothing about distribution: it is concerned only with local (same process) transactions. However, importantly it has sufficient hooks to enable the transactions it creates to be distributed in a manner which makes sense for the environment in which it operates, e.g., CORBA IIOP or SOAP/XML.

The design and implementation goal of *ArjunaCore* was to provide a state of the art programming system for constructing fault-tolerant distributed applications. In meeting this goal, three system properties were considered highly important:

- *Modularity*. The system should be easy to install and run. In particular, it should be possible to replace a component of *ArjunaCore* by an equivalent component already present in the underlying system.

- *Integration of mechanisms*. A fault-tolerant system requires a variety of system functions for concurrency control, error detection and recovery from failures, etc. These mechanisms must be provided in an integrated manner such that their use is easy and natural.

- *Flexibility*. These mechanisms should also be flexible, permitting application specific enhancements, such as type-specific concurrency and recovery control, to be easily produced from the existing default ones.

- *Distribution*. Most of the differences between different distributed transaction systems are typically in the way in which the transaction context is propagated: the core transaction engines vary little between these environments since they are all based on the same two-phase commit protocol. Therefore, is should be relatively easy to embed *ArjunaCore* in any environment and distribute context information appropriately.

In *ArjunaCore,* the first goal has been met by dividing the overall system functionality into a number of modules which interact with each other through well defined *narrow* interfaces. The remaining goals have been met primarily through the provision of a classes which have been organised into a class hierarchy in a manner that will be familiar to developers object-oriented systems. *ArjunaCore* assumes that every major entity in the application is an object. This philosophy also applies to the internal structure of *ArjunaCore* itself. Thus, *ArjunaCore* not only supports an object-oriented model of computation, but its internal structure is also object-oriented. This approach has permitted the use of the *inheritance* mechanisms of object-oriented systems for incorporating the properties of fault-tolerance in a very flexible and integrated manner.

3 Failure Assumptions and Computation Model

3.1 Failure assumptions

It is assumed that the hardware components of the system are computers (nodes), connected by a communication subsystem. A node is assumed to work either as specified or simply to stop working (crash). After a crash, a node is repaired within a finite amount of time and made active again. A node may have both stable (crash-proof) and non-stable (volatile) storage or just non-stable storage. All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage remains unaffected by a crash. Faults in the underlying communication subsystem may result in failures such as lost, duplicated or corrupted messages. Well known network protocol techniques are available for coping with such failures, so their treatment will not be discussed further.

3.2 Objects and transactions

As indicated, we are considering a computation model in which application programs manipulate persistent (long-lived) objects under the control of atomic transactions. Each object is an instance of some class. The class defines the set of *instance variables* each object will contain and the *operations* or *methods* that determine the behaviour of the object. The operations of an object have access to the instance variables and can thus modify the internal state of that object. All operation invocations may be controlled by the use of atomic transactions which have the well known ACID properties of:

- *Atomic*: if interrupted by failure, all effects are undone (rolled back).
- *Consistent*: the effects of a transaction preserve invariant properties.
- *Isolated*: a transaction's intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they are performed concurrently.
- *Durable*: the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure).

Atomic transactions can be nested. A commit protocol is used during the termination of an outermost atomic transaction (*top-level transaction*) to ensure that either all the objects updated within the transaction have their new states recorded on stable storage (committed), or, if the transaction aborts, no updates get recorded. Typical failures causing a computation to be aborted include node crashes and continued loss of messages caused by a partition. It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent (class specific) state changes to the object. Atomic transactions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

The object and transaction model provides a natural framework for designing fault-tolerant systems with persistent objects. When not in use a persistent object is assumed to be held in a *passive* state in an object store (a stable object repository) and is *activated* on demand (i.e., when an invocation is made) by loading its state and methods from the persistent object store to the volatile store, and associating with it an object container.

3.2.1 Commit protocol

A two-phase commit protocol is required to guarantee that all of the transaction participants either commit or abort any changes made. Figure 1 illustrates the main aspects of the commit protocol: during phase 1, the transaction coordinator, C, attempts to communicate with all of the transaction participants, A and B, to determine whether they will commit or abort. An abort reply from any participant acts as a veto, causing the entire transaction to abort. Based upon these (lack of) responses, the coordinator arrives at the decision of whether to commit or abort the transaction. If the transaction will commit, the coordinator records this decision on stable storage, and the protocol enters phase 2, where the coordinator forces the participants to carry out the decision. The coordinator also informs the participants if the transaction aborts.

When each participant receives the coordinator's phase 1 message, they record sufficient information on stable storage to either commit or abort changes made during the transaction. After returning the phase 1 response, each participant which returned a commit response *must* remain blocked until it has

received the coordinator's phase 2 message. Until they receive this message, these resources are unavailable for use by other transactions. If the coordinator fails before delivery of this message, these resources remain blocked. However, if crashed machines eventually recover, crash recovery mechanisms can be employed to unblock the protocol and terminate the transaction.

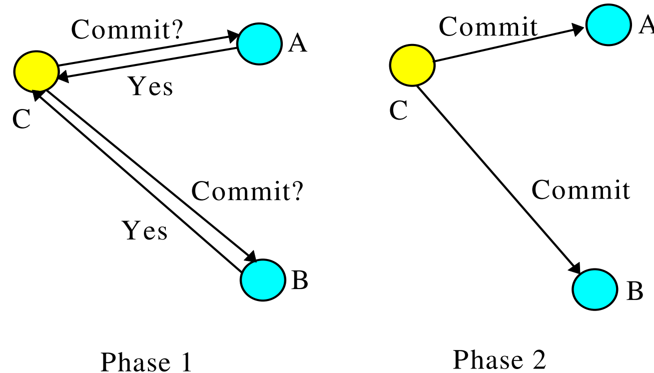

Figure 1: Two-phase commit protocol.

3.3 System Architecture

With the above discussion in mind, we will first present a simple client-server based model for accessing and manipulating persistent objects and then present the overall system architecture necessary for supporting the model. We will consider a system without any support for object replication, deferring the discussion on replication to a later section.

We assume that each persistent object possesses a unique, system given identifier (UID). The passive representation of an object in the object store may differ from its volatile store representation. The *ArjunaCore* model assumes that an *object* is responsible for providing the relevant state transformation operations that enable its state to be stored and retrieved from the object store. Further, we assume that each object is responsible for performing appropriate concurrency control to ensure serialisability of atomic transactions. In effect this means that each object will have a concurrency control object associated with it. In the case of locking, each method will have an operation for acquiring, if necessary, an appropriate lock from the associated lock manager before accessing the object's state; the locks are released when the commit/abort operations are executed.

We can now identify the main modules of *ArjunaCore* and the services they provide for supporting persistent objects.

- (i) *Atomic Transaction module.* Provides atomic transaction support to application programs in the form of operations for starting, committing and aborting transactions;
- (ii) *Object Store module.* Provides a stable storage repository for objects; these objects are assigned unique identifiers (UIDs) for naming them.

The *ArjunaCore* structure is highly modular: by encapsulating the properties of persistence, recoverability, shareability, serialisability and failure atomicity in an Atomic Transaction module and defining narrow, well-defined interfaces to the supporting environment, *ArjunaCore* achieves a significant degree of modularity as well as portability.

We will now use a simple program to illustrate how these modules interact. The program shown below is accessing two existing persistent objects, `A`, an instance of class `O1` and `B`, an instance of class `O2`.

```
{
    O1 objct1(UidA);            /* bind to A */
    O2 objct2(UidB);            /* bind to B */
    AtomicAction act;
    act.begin();                /* start of atomic action act */
    objct1.op(...);
    objct2.op(...);             /* invocations ....*/
    .......
    act.commit();               /* act commits */
 }                              /* break bindings to A and B */
```

Program 1: Outline transaction example.

Thus, to bind to `A`, a local instance of `O1` called `objct1` is created, passing to its constructor an instance of the UID representing its persistent state. The Object Store module of *ArjunaCore* enables the object (or container) to load the latest (committed) state of the object from the object store of a node. The state is loaded, where necessary, as a side effect of locking the object. To manipulate objects under the control of an atomic action, the application creates a new instance of an transaction (`act`) and invokes its `begin` operation. The `commit` operation is responsible for committing the transaction (using a two-phase commit protocol).

4 Implementing the System

The following sub-sections describe in detail how the architecture outlined in the previous sections has been implemented in *ArjunaCore*. We start by concentrating on object storage, retrieval, and the atomic transaction system.

4.1 The Life Cycle of an ArjunaCore Object

A persistent object not in use is assumed to be held in a *passive* state with its state residing in an object store (in *ArjunaCore* this is implemented by the class `ObjectStore`) and *activated* on demand. Passive representations of an object are held as instances of the class `ObjectState`. Such instances are compacted machine and architecture independent forms of arbitrary user-defined objects. As such they can be stored in the object store for persistence purposes; held in memory for recovery purposes; or transmitted over a communications medium for distribution purposes. The class `Input/OutputObjectState` is responsible for maintaining a buffer into which the instance variables that constitute the state of an object may be contiguously saved and provides a full set of operations that allows the runtime representation of a Java object to be converted to and from an `Input/OutputObjectState` instance. The fundamental life cycle of a persistent object in *ArjunaCore* is shown in Figure 3.

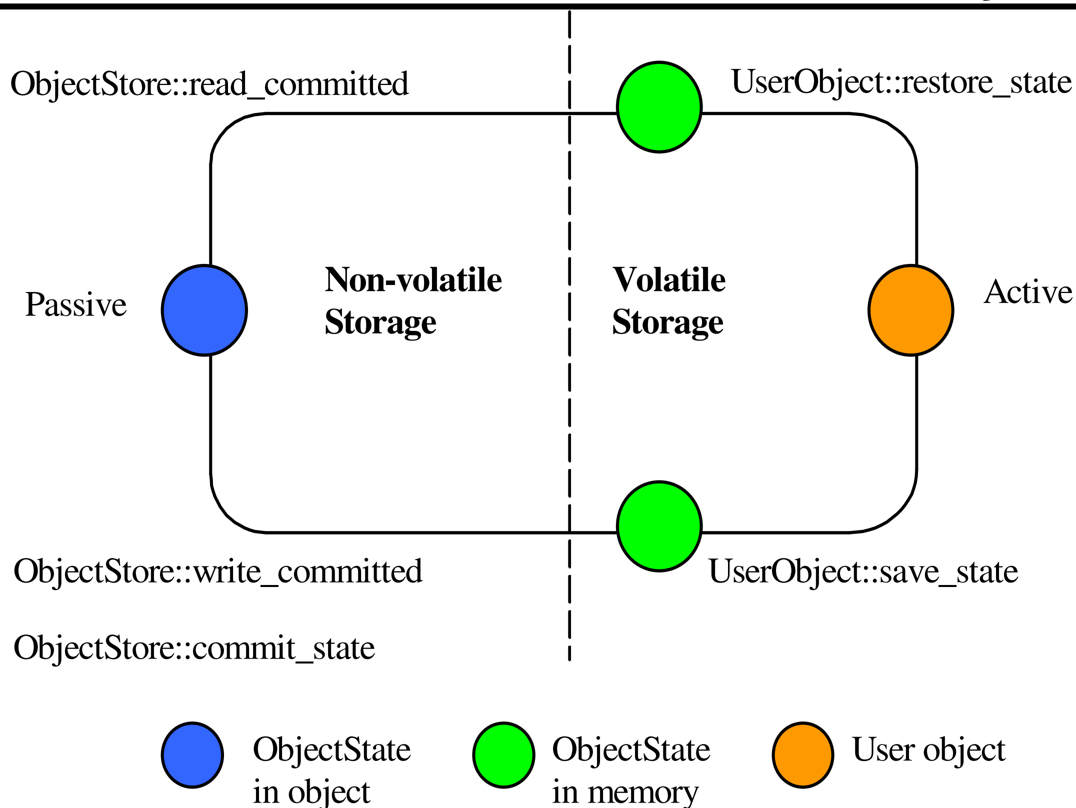

Figure 3: The lifecycle of a persistent object.

- The object is initially passive, and is stored in the object store as an instance of the class OutputObjectState.
- When required by an application the object is automatically activated by reading it from the store using a read_committed operation and is then converted from an InputObjectState instance into a fully-fledged object by the restore_state operation of the object.
- When the application has finished with the object it is deactivated by converting it back into an OutputObjectState instance using the save_state operation, and is then stored back into the object store as a shadow copy using write_uncommitted. This shadow copy can be committed, overwriting the previous version, using the commit_state operation. The existence of shadow copies is normally hidden from the programmer by the atomic action system. Object de-activation normally only occurs when the top-level action within which the object was activated commits.

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

The operations `save_state` and `restore_state` are fundamental operations that form part of the interface provided by the class `StateManager`. Their definition and use are described in a later section.

4.2 Implementing Object Storage Services

4.2.1 Saving object states

ArjunaCore 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), persistence (the state represents the final state of an object at application termination), and for distribution purposes (the state represents the current state of an object that must be shipped to a remote site). Since all of these requirements require common functionality they are all implemented using the same mechanism - the classes `Input/OutputObjectState` and `Input/OutputBuffer`.

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

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

    /* pack operations for standard Java types */

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

public class InputBuffer
{
public InputBuffer ();

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

    /* unpack operations for standard Java types */

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

The `Input/OutputBuffer` class maintains an internal array into which instances of the standard Java types can be contiguously packed (unpacked) using the `pack (unpack)` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form (so-called network byte order) to make them machine independent.

```
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 class `Input/OutputObjectState` provides all the functionality of `Input/OutputBuffer` (through inheritance) but adds two additional instance variables that signify the `Uid` and `type` of the object for which the `Input/OutputObjectState` instance is a compressed image. These are used when accessing the object store during storage and retrieval of the object state.

4.2.2 The Object Store

The object store provided with *ArjunaCore* 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. *Note*: as with all *ArjunaCore* classes the default object stores are pure Java implementations; to access the shared memory and other more complex object store implementations it is necessary to use native methods.

All of the object stores hold and retrieves instances of the class `Input/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 *ArjunaCore* system components as appropriate thus the existence of any shadow versions of objects in the store are hidden from the programmer.

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

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

When a transactional object is committing it is necessary for it to make certain state changes persistent in order that it can recover in the event of a failure and either continue to commit, or rollback. To guarantee ACID properties, these state changes must be *flushed* to the persistence store implementation before the transaction can proceed to commit; if they are not, 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, *ArjunaCore* ensures that such state changes are flushed. However, doing so can impose a significant performance penalty on the application.
In order to improve the performance of the store several optimisations are implemented. The first is an open file cache that keeps files containing object states open as long as possible. This is to reduce the considerable overhead UNIX imposes for file system opens. Files are automatically added to this cache when first used and remain in it until they are explicitly removed or the cache needs compacting. The cache size is configurable and is initially set to use 50% of the available file descriptors for a process. The second optimisation overcomes read latency introduced by the use of a header. When a shadow copy is committed the store determines if the object is small enough to fit into the disk block reserved for the header. If it will then it is written to that block immediately after the control information as part of the same write system call that replaces the header block. Thus when the header is later read the last committed state may also be implicitly read and is immediately available without the need to read either of the object storage areas.

4.3 Implementing Atomic Transaction Services

4.3.1 Overview

The principal classes which make up the class hierarchy of the *ArjunaCore* Atomic Transaction module are depicted below.

```
StateManager                // Basic naming, persistence and recovery control
    LockManager             // Basic two-phase locking concurrency control
        User-Defined Classes
    Lock                    // Standard lock type for multiple readers/single writer
        User-Defined Lock Classes
    AtomicAction            // Implements atomic transaction control abstraction
    AbstractRecord          // Important utility class
        RecoveryRecord      // handles object recovery
        LockRecord          // handles object locking
        RecordList          // Intentions list
        other management record types
```

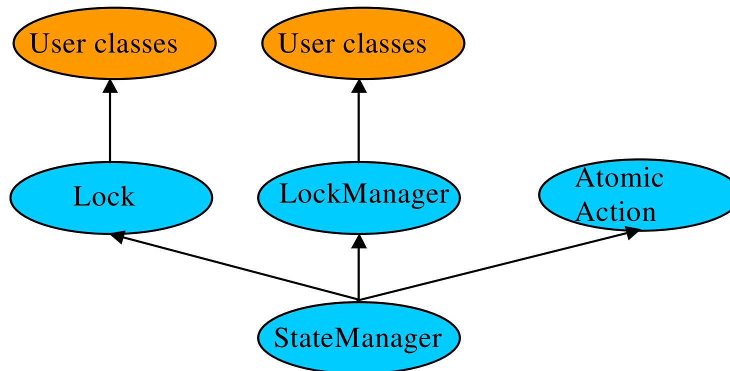

Figure 2: ArjunaCore class hierarchy

To make use of atomic transactions in an application, instances of the class `AtomicAction` must be declared by the programmer in the application as illustrated earlier. The operations this class provides (`begin`, `abort`, `commit`) can then be used to start and manipulate atomic transactions (including nested transactions). The only objects controlled by the resulting transactions are those objects which are either instances of *ArjunaCore* classes or are user-defined classes derived from `LockManager` and hence are members of the hierarchy shown above.

Most *ArjunaCore* system 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. Thus, instances of the class `StateManager` are the principal users of the object store service. The class `LockManager` uses the facilities of `StateManager` and provides the concurrency control (two-phase locking in the current implementation) required for implementing the serialisability property of atomic transactions. The implementation of transaction facilities for recovery, persistence management and concurrency control is supported by a collection of object classes derived from the class `AbstractRecord` which is in turn derived from `StateManager`. For example, instances of `LockRecord` and `RecoveryRecord` record recovery information for `Lock` and user-defined objects respectively. The `AtomicAction` class manages instances of these classes (using an instance of the class `RecordList` which corresponds to the *intentions list* used in traditional transaction systems) and is responsible for performing aborts and commits.

Consider a simple example. Assume that `Example` is a user-defined persistent class suitably derived from the *ArjunaCore* class `LockManager`. An application containing a transaction `Trans` accesses an object (called `O`) of type `Example` by invoking the operation `op1` which involves state changes to `O`. The serialisability property requires that a write lock must be acquired on `O` before it is modified; thus the body of `op1` should contain a call to the `setlock` operation of the concurrency controller:

```
Boolean Example::op1 (...)
{
    if (setlock (new Lock(WRITE)) == GRANTED)
    {
        // actual state change operations follow
        ...
    }
}
```

Program 4: Simple concurrency control

The operation `setlock`, provided by the `LockManager` class, performs the following functions in this case:

- Check write lock compatibility with the currently held locks, and if allowed:
- Call the `StateManager` operation `activate` that 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) and inserting it into the `RecordList` of `Trans`.
- Create and insert a `LockRecord` instance in the `RecordList` of `Trans`.

Now suppose that transaction `Trans` is aborted sometime after the lock has been acquired. Then the `abort` 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/PersistenceRecord` will restore the prior state of `O`.

Each of these classes and their relationships with each other will be described in greater detail in the following sections.

4.3.2 Recovery and Persistence

At the root of the class hierarchy in *ArjunaCore* is the class `StateManager`. As indicated before, this class is responsible for object activation and deactivation, object recovery and also maintains object names (in the form of object UIDs).

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

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

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

public synchronized void destroy ();

public final Uid get_uid ();

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

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

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

Program 5: class StateManager

Objects are assumed to be of three possible basic flavours. They may simply be *recoverable* (signified by the constructor argument `RECOVERABLE`), in which case `StateManager` will attempt to generate and maintain appropriate recovery information for the object (as instances of the class `Input/OutputObjectState` as mentioned earlier). Such objects have lifetimes that do not exceed the application program that creates them. Objects may be *recoverable and persistent* (signified by `ANDPERSISTENT`), in which case 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` will attempt to automatically load (unload) any existing persistent state for the object by calling the `activate` (`deactivate`) operation at appropriate times. Finally, objects may possess none of these capabilities (signified by NEITHER) in which case no recovery information is ever kept nor is object activation/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. This restriction simplifies some aspects of the overall object management.

If an object is recoverable (or persistent) then `StateManager` will invoke the operations `save_state` (while performing `deactivation`), `restore_state` (while performing `activate`) and `type` at various points during the execution of the application. These operations *must* be implemented by the programmer. However, the capabilities provided by `Input/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 the following:

```
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;
    }
}
```

Program 6: Example save_state Code

Since `StateManager` cannot detect user level state changes, it also exports an operation called `modified`. It is the responsibility of the programmer to call this operation prior to making any changes in the state of an object (as discussed before, this is normally automatically done via the concurrency controller).

The `get_uid` operation 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 (for existing objects) or by generating a new identifier when the object is created.

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. That is, it uses instances of the class `Input/OutputObjectState` both for recovery and persistence purposes. An additional argument passed to the `save_state` and `restore_state` operations allows the programmer to determine the purpose for which any given invocation is being made thus allowing different information to be saved for recovery and persistence purposes.

4.3.3 The Concurrency Controller

The concurrency controller is implemented by the class `LockManager` (Program 7) which provides sensible default behaviour 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 *ArjunaCore* runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. 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 defaultTimeout;
public static final int defaultRetry;
public static final int waitTotalTimeout;

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

     /* abstract methods inherited from StateManager */

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

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

Program 7: class LockManager

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.

4.3.4 Locking Policy

Unlike many other systems, locks in *ArjunaCore* are not special system types. Instead they are simply instances of other *ArjunaCore* 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 `Lock` class instances which provide

operations (the `conflictsWith` operation) by which `LockManager` can determine if two locks conflict or not.

```
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 ();
    . . .
};
```

Program 8: class Lock

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. 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 default `Lock` class supports the traditional multiple reader/single writer policy.

4.3.5 Co-ordinating Recovery, Persistence and Concurrency Control

Since objects are assumed to be encapsulated entities then they must be responsible for implementing the properties required by atomic actions themselves (with appropriate system support). This enables differing objects to have differing recovery and concurrency control strategies. Given this proviso then any atomic action implementation need only control the invocation of the operations providing these properties at the appropriate time and need not know how the properties themselves are actually implemented.

```
public class BasicAction extends StateManager
{
public BasicAction ();
public BasicAction (Uid objUid);

public final int status ();

public final boolean preventCommit ();

public final synchronized int add (AbstractRecord rec);

public final BasicAction parent ();

public final int typeOfAction ();

public final Uid topLevelActionUid ();
public final BasicAction topLevelAction ();

public final synchronized CheckedAction setCheckedAction (CheckedAction act);

public static BasicAction Current ();

protected synchronized int Begin (BasicAction parent);
protected synchronized int End (boolean report_heuristics);
protected synchronized int Abort ();

protected synchronized final int prepare (boolean report_heuristics);

protected synchronized final void phase2Commit (boolean report_heuristics);
protected synchronized final void phase2Abort (boolean report_heuristics);

protected RecordList pendingList;
protected RecordList preparedList;
protected RecordList readonlyList;
protected RecordList failedList;
protected RecordList heuristicList;
};
```

class BasicAction

```
public class AtomicAction extends BasicAction
{
public AtomicAction ();
public AtomicAction (Uid objUid);

public synchronized int begin ();
public int commit ();
public synchronized int commit (boolean report_heuristics);
public synchronized int abort ();

public String type ();

public boolean addThread ();
public synchronized boolean addThread (Thread t);
public boolean removeThread ();
public synchronized boolean removeThread (Thread t);
};
```

Program 9: class AtomicAction

In order to accomplish this `AtomicAction` instances maintain a list of instances of classes derived from a special abstract management class called `AbstractRecord`. Each of these classes manages a certain property, thus `RecoveryRecords` manage object recovery; `LockRecords` manage concurrency control information, etc. Instances of these management records are automatically added to the `pendingList` of the current atomic action as appropriate during execution of the application. Given this list of management records then it is thus sufficient for the operations of `AtomicAction` to run down the list invoking an appropriate operation on each record instance.

```
public abstract class AbstractRecord extends StateManager
{
public abstract int typeIs ();

public abstract Object value ();
public abstract void setValue (Object o);

public ClassName className ();

public abstract int nestedAbort ();
public abstract int nestedCommit ();
public abstract int nestedPrepare ();

public abstract int topLevelAbort ()
public abstract int topLevelCommit ();
public abstract int topLevelPrepare ();

public Uid order ();
public String getTypeOfObject ();

public boolean propagateOnAbort ():
public boolean propaagteOnCommit ();

public boolean equals (AbstractRecord rec);
public boolean lessThan (AbstractRecord rec);
public boolean greaterThan (AbstractRecord rec);
};
```

Program 10: class AbstractRecord

Thus, when an action is committed by the user (using the `commit` operation) then the two phase protocol implemented by `AtomicAction` is performed. This consists of firstly invoking the appropriate prepare phase operation (`topLevelPrepare` or `nestedPrepare`) on each of the records held in the `pendingList`. As each record is processed it is moved from the `pendingList` to either the `preparedList` or the `readonlyList` depending upon whether the record needs take part in phase two of the commit protocol. Each such invocation returns a status indicating whether the operation succeeded or not. If any failures are detected the prepare phase is terminated and the action will be aborted in phase two of the protocol.

Once the prepare phase has terminated `AtomicAction` will either invoke `phase2commit` or `phase2Abort` depending upon the result of the prepare phase. If the prepare phase for a top level action completes successfully (indicating that the action should be committed) then the state of the atomic action is written to the object store (using the same persistence mechanisms described previously) to ensure that the commit will succeed even if a node crash occurs during phase two of the protocol. Both of these operations are essentially identical in that they process the records held on all of the lists and invoke the appropriate management operation (`topLevelCommit`, `topLevelAbort`, `nestedCommit`, or `nestedAbort`). At this point the records may either be discarded (if the action aborts or is top level) or propagated to the parent action for possibly further processing. For top level actions successful completion of phase two cause the state saved in the object store at the end of the prepare phase to be deleted.

This record based approach provides complete flexibility in that new record types can be created as required (other record types currently handle persistence (`PersistenceRecord`) and object lifetime (`ActivationRecord`)).

As a demonstration of the simplicity of using actions in *ArjunaCore*, the following class represents the interface to a simple distributed diary system:

```
public class Diary extends LockManager
{
public Diary (Uid objUid);

public String WhereIs (int time, String user);

public AnAppointment GetNextAppointment (int now);
public int AddAppointment (AnAppointment entry);
public int DelAppointment (int when);

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

private String user_name;
private AppointmentList appts;
};
```

Program 11: class Diary

The `GetNextAppointment` operation of this class could be written as shown in Program 12. The *ArjunaCore* additions to this code consist simply of the declaration and use of an instance of the `AtomicAction` class and the insertion of a call to the inherited `setlock` operation. The remainder of the code is exactly as it would be had *ArjunaCore* not been used. In this case read locks are set to ensure that the list of appointments is not modified during the search for the next valid appointment. These locks are automatically released (or propagated to a parent action if one exists) if the action commits or they will be released regardless if the action aborts.

```
public AnAppointment GetNextAppointment( int time )
{
     AtomicAction A = new AtomicAction();
     AnAppointment entry = null;

     A.begin();
     if (setlock(new Lock(LockMode.READ), 0) == LockResult.GRANTED)
     {
          AppointmentList tmp = appts;

          entry.start = 0;
          entry.end = 0;
          entry.description = "";

          while (tmp != null)
          {
               if (tmp.entry.start <= time)
                    tmp = tmp.next;
               else
               { // found first appointment starting after given time
                    entry = tmp.entry;
                    break;
               }
          }
          A.commit();
     }
     else
          A.abort();
     return entry;
}
```

Program 12: Example User Code.

4.4 Crash Recovery

The *ArjunaCore* crash recovery mechanism is built around the fact that when a top level transaction that has modified the states of some objects prepares successfully it saves its state in the object store (pure read-only actions do not need to do this). Thus, by examining the contents of the object store the crash recovery mechanisms can determine those actions that had prepared but not fully committed.

The crash recovery process consists of scanning the object store for saved transaction states (that is transactions that had prepared but not committed or aborted). Once the commit or abort decision has been completed the transaction entry is deleted from the object store.

The query process at the co-ordinating node uses the presence (or absence) of a corresponding atomic transaction entry in the object store to determine whether the transaction committed or aborted. The system works in a *presumed abort* mode, that is, if no record exists then the transaction is aborted. This arises from the fact that the record is written only at the end of a successful prepare phase implying that all participants have agreed to commit. If any cannot commit then the record will not be written. Similarly, the record is deleted only when phase two has been successfully completed, in which case there cannot be any transactions whose state is unknown.