The Treatment of Persistent Objects in Arjuna

G.N. DIXON, G.D. PARRINGTON, S.K. SHRIVASTAVA
and S.M. WHEATER

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK.

ABSTRACT

Arjuna is a programming system which provides a set of tools for constructing fault-tolerant distributed applications. It supports an object-oriented model of computation in which atomic actions (atomic transactions) control sequences of operations invoked upon persistent objects. Persistent objects outlive the applications that create them and this paper concentrates on the mechanisms within *Arjuna* that are concerned with their management. The paper describes how these mechanisms are related to the other *Arjuna* mechanisms required by atomic actions for the distribution, concurrency control, recovery and commitment of persistent objects.

1 INTRODUCTION

Arjuna is an object-oriented programming system that permits the creation of fault-tolerant, distributed applications. A prototype version has been designed and implemented in C++ [Stroustrup 86] to run on a set of UNIX[†] workstations connected by a local area network. *Arjuna* provides nested atomic actions (also known as nested atomic transactions [Gray 78]) for structuring applications. An *Arjuna* application typically consists of one or more atomic actions which have been created to control a collection of operations on *Arjuna* objects. An *Arjuna* object is an instance of an abstract data type (a C++ class), which can be made a long-lived (*persistent*) entity that will continue to exist after the application that created it terminates. By ensuring that objects are only manipulated from within atomic actions, it can be guaranteed that the integrity of objects (and hence that of the system) is maintained in the presence of failures such as workstation crashes and loss of network messages.

Any distributed *object and action* based system must provide a number of *integrated* mechanisms for supporting a variety of system functions. These

---

mechanisms include those for naming, locating and invoking operations on (local and remote) objects, concurrency control, recovery control, managing object states for long term as well as short term storage etc. In addition they should be *flexible*, permitting application specific enhancements of existing mechanisms, for example type-specific concurrency control. *Arjuna* has been designed to provide such a set of mechanisms through a number of C++ classes which are organised in a class/type hierarchy that will be familiar to the developers of 'traditional' (single node) centralised object-oriented systems.

This paper concentrates on the treatment of persistent objects within *Arjuna*. The next section describes the overall architecture of *Arjuna*; section 3 then covers the topic of the management of persistent objects including their naming, storage, activation and deactivation. Section 4 presents a simple example - a persistent spreadsheet - to illustrate the main ideas of the paper. The paper concludes by comparing our work with other efforts reported in the literature and discusses several possible extensions, some of which will be incorporated in the next version of *Arjuna*.

2 ARJUNA ARCHITECTURE

This section contains an overview of the *Arjuna* architecture, starting with its computational model. This overview is essential to appreciate the details of the persistent object management mechanisms presented in the next section.

2.1 Objects and Actions

A computational model that has been widely advocated for constructing robust distributed systems is based upon the concept of structuring applications using atomic actions which 'operate' upon objects [Herbert and Monk 87]. An object is an instance of a *type* or *class*. Each individual object consists of some state (its instance variables) and a set of operations (its methods) that determine the externally visible behaviour of the object. The operations supported by a class have access to the instance variables and can thus modify the state of an object. It is assumed that the invocation of any of these operations will produce consistent (class specific) state changes to the objects in the system in the absence of failures.

In a distributed system, the objects needed by an application may be remote from the node where the application is executing, requiring a network protocol that enables an application to invoke an operation on a remote object. Typically, this network protocol is implemented by a *remote procedure call* (RPC) mechanism that passes value parameters from the (*client*) application to the *server*, created to manage the object on the remote node, and returns the result of the operation's execution. All operation invocations may be controlled using *atomic actions* which have the properties of (i) *serialisability*, (ii) *failure atomicity*, and (iii) *permanence of effect*. The first property ensures that concurrent execution of programs which require access to common objects are free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution [Eswaran 76]). The second property ensures

that a computation either terminates normally (*committed*) producing the intended results, or it can be *aborted* producing no results. Whenever a failure that cannot be masked occurs, an atomic action is aborted. Typical failures causing a computation to be aborted include node (workstation) crashes and communication failures such as the repeated loss of messages. It is reasonable to assume that once a computation terminates normally, the results produced are not destroyed by subsequent node crashes. This is the third property - permanence of effect - which ensures that state changes produced (i.e. new states of persistent objects modified by the action) are recorded on *stable storage*, a type of storage which can survive node crashes with high probability. A *commit protocol* is required during the termination of an action to ensure that either all the objects updated within the action have their new states recorded on stable storage (normal termination), or no updates get recorded (aborted termination) [Gray 78]. Some form of concurrency control policy, such as that enforced by two phase locking, is also required to ensure the serialisability property of actions [Eswaran 76].

The *object and action* model provides a natural framework for designing fault-tolerant systems with persistent objects. Persistent objects normally reside in object stores which are designed to be stable. Atomic actions are employed to control the state changes made to these objects and the properties given above for atomic actions ensure that only consistent state transformations take place on objects, despite failures. Normally an object resides on a single node (and hence in a single object store), however the availability of an object can be increased by replicating it on different nodes, and as a result storing copies in more than one object store. Such replicated objects must be managed through appropriate *replica-consistency protocols* to ensure that the object copies remain mutually consistent. Protocols of this type can be integrated within action based systems (for example, by enhancing the commit protocol) as discussed in [Bernstein *et al.* 87].

It is assumed that the hardware components of the system are workstations (nodes), connected by a communication subsystem (for example, a local area network). 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 can have both stable and non-stable (*volatile*) storage or just non-stable storage (the former modelling a diskfull workstation and the other a diskless one). All the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage, as stated earlier, remains unaffected by a crash. It is also assumed that the faults in the communication subsystem are responsible for failures such as lost, duplicated or corrupted messages. Well known network protocol level techniques are available for coping with such failures, so their treatment will not be discussed further.

2.2 System Architecture

Assume that the system is quiescent with no applications executing, so that all of the objects in the system are in a *passive* state and stored in object stores. An object becomes *active* once an operation is invoked upon it from within an atomic action. When an atomic action terminates, any objects that were activated within that atomic action will remain active provided that an enclosing atomic action exists, otherwise all modified objects will be deactivated during commit processing.

To further explain the architectural features of *Arjuna* consider the system which consists of six nodes shown in Figure 1. In this system, one node (N_4) is

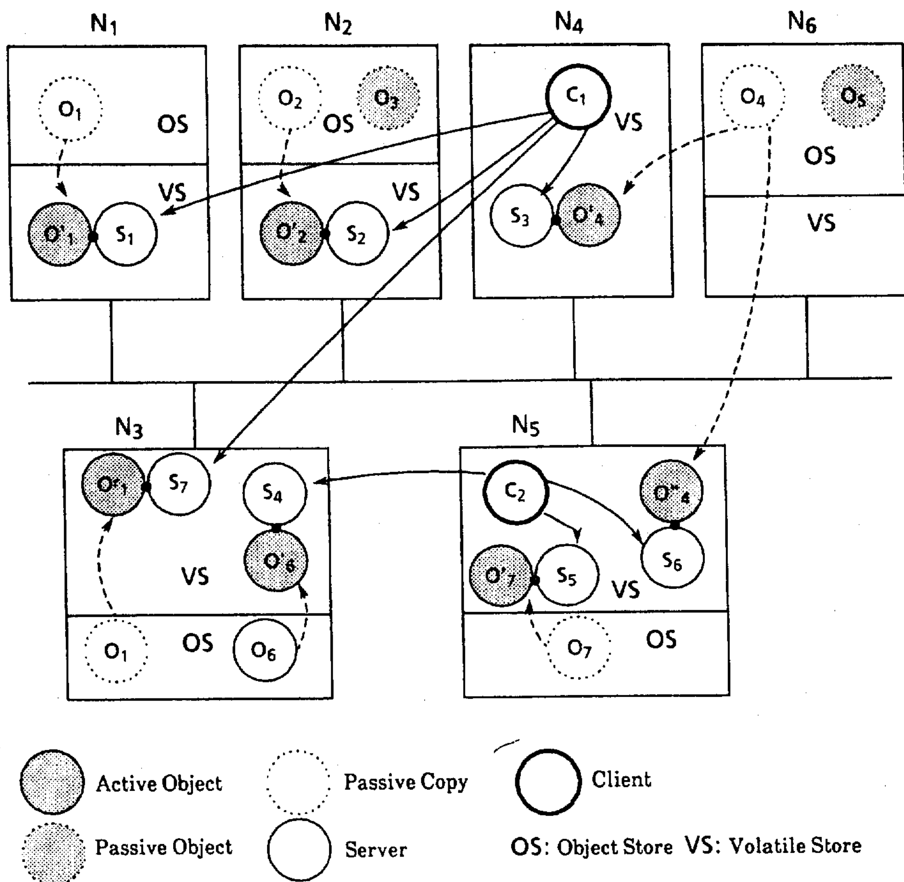

Figure.1: Objects, Clients and Servers

diskless, and others have disks organised as object stores. An atomic action which is part of the application being executed by client process C_1 *(at* N_4*) is*

controlling access to objects O_1 (at N_1), O_2 (at N_2) and O_4 (at N_6). These objects have been activated; for example O'_2 is an active object, which is being managed by server S_2 on behalf of C_1; O_2 represents the old passive copy of the object in the object store. Normally persistent objects are activated at the node where their passive state is held, however, an object can also be activated at an alternative node (for example, O_4 has been activated at N_4). This *remote activation* mechanism provides a simple object caching scheme for fast access. Objects can also be *replicated*, for example in the figure object O_1 is replicated with a copy at N_3. The activation scheme for replicated objects will depend upon the replication strategy being employed for that object. For example, in the *available copy* approach [Bernstein *et al.* 87], all the available copies (i.e. the ones on operating nodes) will be activated (as shown in the figure, where $O^r{}_1$ is the active replica of O'_1). The figure also shows another client (C_2) accessing a number of objects. Both C_1 and C_2 have access to O_4. Multiple activations of an object are permitted provided no state changes are involved (this corresponds to the 'shared read' access). Both C_1 and C_2 have cached O_4 at their local hosts. The execution of an operation may involve the invocation of operations on remote objects, resulting in a client-server hierarchy of arbitrary depth as servers also act as clients; for the sake of simplicity such a deep hierarchy has not been depicted in the figure. When the client application terminates, the top-level client process (C_1 in this example) will initiate the commit of the action in which all the servers (S_1, S_2, S_3 and S_7 for C_1) must take part.

Implementation details of the mechanisms that support persistence will be presented in the next subsection and in section 3, but from the discussion given above, some of the main requirements can be appreciated. A consistent scheme for naming various entities (for example, hosts, object stores, objects, classes, servers, etc.) and a binding mechanism for binding a given named entity to its current address is required; in *Arjuna* this functionality is provided - as in many other distributed systems [Oppen and Dalal 81] - by one or more *name server* objects. Since activation (deactivation) of an object involves moving the object from (to) the object store to (from) main memory, possibly through a network, mechanisms for changing an object's representation are also required. When objects are manipulated from within atomic actions, appropriate recovery data (prior states of objects) must also be maintained in case an action has to be aborted.

2.3 Implementation

This subsection contains some relevant information on the overall implementation of the system. As stated earlier, an operation on an object is invoked via an RPC. At the application level, objects are the only visible entities; the client and server processes that do the actual work are hidden. In *Arjuna*, server processes are created dynamically as RPCs are made to objects by the RPC system *Rajdoot* [Panzieri and Shrivastava 88], which has been considerably enhanced to provide support for object group management. Supporting replication is the principal use of the group management operations,

however additional use is made by the atomic action mechanisms to control action termination by treating all servers created during an atomic action as a single group. *Rajdoot* also detects and exterminates *orphans* - computations that occur when the controlling client program terminates prematurely. For example, referring to the scenario depicted in Figure 1, if node N_4 crashes, then the servers on remote nodes (S_1, S_2, S_7) will become orphans and will be automatically killed (after suitable recovery has taken place). A C++ stub generator has been implemented to provide a high level interface to the RPC system. The classes produced by the stub generator automate the task of binding object names to locations, server management and parameter marshalling.

The class hierarchy of *Arjuna* is depicted in Figure 2. To use atomic actions in

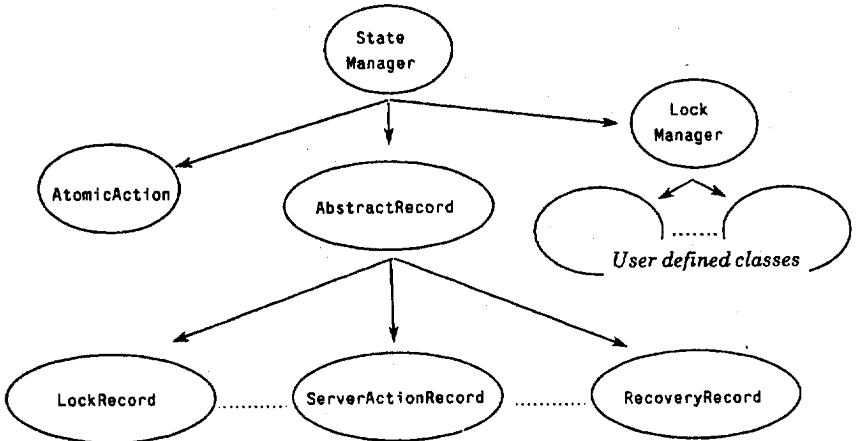

Figure 2: The *Arjuna* class hierarchy

an application, instances of the class AtomicAction must be declared in the application; the operations this class provides (Begin, Abort, End) can then be used in a suitable manner (see section 4 for an example). The only objects that are controlled by the resulting atomic actions are those objects which are either instances of *Arjuna* classes or are user defined classes derived from the class LockManager as shown in Figure 2.

Arjuna is novel with respect to other fault-tolerant distributed systems (for example, Clouds [Dasgupta *et al.* 85] or Argus [Liskov 88]) in taking the approach that every major entity in the system is an object. This approach permits the use of the type inheritance mechanism for incorporating the functionality necessary for concurrency control, recovery, and persistence as required by the object and action model discussed earlier. Thus, all *Arjuna* classes are derived from the base class StateManager, which provides the primitive facilities necessary for constructing persistent objects and atomic actions. These facilities include support for object activation and deactivation,

along with state-based recovery. The class LockManager uses these facilities and provides support for concurrency control - (two phase locking in the current implementation) - which is required to support the serialisability property of atomic actions. The (*record*) classes derived from AbstractRecord provide facilities for recording action management information for specific properties of an object; for example, LockRecord maintains management information for Lock objects. The AtomicAction class manages instances of these classes (using an instance of the class RecordList) and is responsible for performing aborts and commits using the operations the *record* classes support.

As an example, assume that O is a user-defined persistent object (an instance of the class MailBox which is currently passive) and an application containing atomic action A accesses this object by invoking an operation insert which involves state changes. Thus, in order to satisfy the serialisability requirements, a write lock must be acquired on O before O is modified (assuming a two-phase locking approach is being used). The body of operation insert should then contain a call to the appropriate locking operation:

```
MailBox::insert (...)
{
    setlock (new Lock(WRITE));
    // actual state change operations follow ...
    ...
}
```

The operation setlock is provided by the LockManager class, and performs the following functions:- (i) check if O can be locked in WRITE mode; (ii) if so, activate O by using the StateManager operation activate; (iii) call the modified operation of StateManager (since the Lock is a WRITE lock) which results in the creation of a RecoveryRecord containing a snapshot of the current state of O and its insertion into the RecordList of A; (iv) create and insert a LockRecord instance in the RecordList of A.

Suppose now that action A is to be aborted, then the Abort operation of AtomicAction will process the RecordList associated with instance A, invoking the abort operation of the various record instances held (RecoveryRecord, LockRecord, etc.). In the case of the LockRecord instance held by A, the invocation of this operation will release the WRITE lock, while for the RecoveryRecord instance the prior state of O will be restored.

The overall architecture of *Arjuna* together with its design and implementation are discussed in [Shrivastava *et al.* 89]. Object state management for recovery and persistence are discussed in [Dixon and Shrivastava 87, Dixon 88] while principles behind concurrency control in *Arjuna* are discussed in [Parrington and Shrivastava 88, Parrington 88]. At the time of writing, most aspects of *Arjuna have been implemented. Object caching and the naming service have not yet been completely implemented and object replication is not yet fully*

operational. Nevertheless, application programs composed out of nested (and concurrent) atomic actions operating on persistent objects can be developed and tested using simple static binding of names.

3 PERSISTENCE

The design and implementation of the mechanisms which support persistence must address issues such as how to name an existing (persistent) object, from both a user and system-level point of view, and how to arrange that the state of an object is stored in a stable storage medium between activations of the object by applications. This section addresses these issues, and the solutions adopted by the *Arjuna* programming system, beginning with object naming. The following subsection describes the construction of persistent objects, and how the persistence of an object can be ensured by saving the state of the object in an object store. The final subsection describes the design and implementation of the object store provided by *Arjuna*.

3.1 Object naming

In any system that supports some form of persistence, a mechanism is needed to enable a user of that system to name, and subsequently access, an existing persistent resource. In conventional operating systems (such as UNIX), the persistent resources most commonly accessed are files which are named at the user level by a string that identifies the file in the file system supported by the underlying operating system. In *Arjuna* the persistent resources are instances of classes that have been constructed using tools provided by *Arjuna*. Hence, mechanisms are needed to allow the user of a persistent object to name the object in such a way that the persistence mechanisms provided by *Arjuna* can locate the passive form of the object in the object store.

In addition to the user level naming scheme, the system requires a uniform method for naming resources. This system level naming scheme can then be used by the mechanisms that support the persistence of resources. Clearly a mapping must exist between the two naming schemes. A commonly used approach at the system level is to make use of identifiers that are guaranteed to be unique throughout the system.

As noted above, most common naming schemes supported by operating systems provide a user with the ability to name a resource via a string. To provide an acceptable user interface to an application that accesses persistent objects, a similar scheme may be provided for persistent objects, along with a mapping from this name to the unique identifier (*uid*) used by the system.

To increase the user name-space, the string employed by a user may be further qualified by the context within which the string is used. Since objects are instances of a class, the name of the class of an object may be used. In addition, when an object is remote from the application which is accessing the object then *the name of the node where the object is located can be employed. Hence, the fully-qualified name of an object can be considered to consist of three parts;*

instance name, class name and node name. However, users need not necessarily be aware of this fully-qualified name if a name server can map partially qualified names supplied by a user to an equivalent fully-qualified name.

The object naming scheme adopted by *Arjuna* is based on the three-part naming scheme described above, which enables the persistence and distribution mechanisms to locate an object within a network of nodes. For the user, *Arjuna* provides a class called ArjunaName which supports a string based naming scheme that may be used to specify some or all of the parts of an object's name. Declaring an instance of ArjunaName and using it as an argument to the actual object instance declaration is sufficient to identify an existing persistent object. For example;

```
SpreadSheet Entertainment(new ArjunaName("SlushFund"));
```

names an existing instance of the class SpreadSheet called SlushFund that will be accessed via the identifier Entertainment in the application. If the persistent object does not exist then it is created at the node where the application is executing. If multiple objects exist on different nodes with the same name then any one may be selected. By qualifying the name further however, specific persistent objects can be accessed, or created on alterative nodes. For example, to access an instance of the SpreadSheet class on a node called snipe the following may be used:

```
SpreadSheet Entertainment(new ArjunaName("SlushFund", "snipe"));
```

In all cases, the mapping mechanism (implemented by the class ArjunaName) relies on access to a name server that maintains all of the required information needed by ArjunaName to locate the object. When the ArjunaName instance is accessed to determine where a server for an object should be created, the ArjunaName object employs the information passed by the user to determine whether there is an entry for the object in the name server, filling in those fields of the ArjunaName object that were not specified by the user.

3.2 Constructing persistent objects

Section 2 briefly described how the lifetime of a persistent object can be considered to be a series of transitions from active to passive object forms. In persistent programming systems, the movement of an object between active and passive forms is managed by the persistence mechanisms, and should be transparent to the user of a persistent object. In addition, when *orthogonal persistence* [Atkinson *et al.* 83] is supported, the property of persistence is independent of both the class of an object and the manner in which the object is used. Many approaches to supporting persistence have concentrated on new languages or extensions to existing languages. The rest of this subsection describes how the persistence property may be supported using the inheritance property of object-oriented languages.

Since persistent objects are manipulated from within atomic actions, these objects must also be made *recoverable* so that their prior states can be restored should an atomic action abort. Recoverability and persistence mechanisms can be viewed as requiring similar functionality, the difference being where the mechanisms maintain the state of an object and when they are used.

To support recoverability, one approach (the default mechanism provided by *Arjuna*) is to take a snapshot of the state of an object before it is modified for the first time within the scope of an atomic action. If the atomic action aborts then the old state can simply replace the new, thereby achieving recovery. Persistence has complementary requirements, in that the new state of the object is used to replace the old state held in the object store at commit time. Hence, if mechanisms are provided that enable the state of an object to be collected in such a way that it can be transferred to or from stable storage, then support for both persistence and recovery is possible.

An implementation that supports both recoverability and persistence is provided by *Arjuna* in the form of a class called StateManager (see the class hierarchy in Figure 2). This class provides a *recoverable interface* (two operations - save_state and restore_state) that operates in conjunction with a class called ObjectState, and includes a unique identifier object which is an instance of the class Uid (see Figure 3). Hence, all classes derived from

```
class StateManager
{
    Uid object_uid;
    ....
protected:
    void modified( );
    ....
public:
    StateManager( );
    ~StateManager( );

    virtual void save_state(ObjectState*);
    virtual void restore_state(ObjectState*);

    boolean activate( );
    boolean deactivate( );

    Uid get_Uid( );
    ...
};
```

Figure 3: The class StateManager

StateManager will inherit the recoverable interface and an operation (called get_Uid) that returns a copy of the object's unique identifier. As a result, all persistent objects which are instances of classes constructed using StateManager *may be named in a uniform manner using unique identifiers.*

The class ObjectState is responsible for maintaining a buffer into which the instance variables that constitute the state of an object may be contiguously saved. As the compiler for the implementation language chosen by *Arjuna* (C++) provides no information about the primitive types that constitute the instance variables of a class, this approach to adding a recoverable interface relies on the implementor of a new (derived) class refining the inherited operations to operate in a manner specific to that class. If such information were available (e.g. the clsTypes field provided by the Objective-C compiler [Cox 86]) then generic routines could be provided to manage all derived classes.

To ease the burden of the implementor of a new *Arjuna* class, the class ObjectState provides a set of operations for most of the primitive types supported by the implementation language (called pack and unpack). The implementation of the operations that constitute the recoverable interface will therefore consist of the invocation of a number of the operations provided by ObjectState.

An an example consider a class that provides the abstraction of a mailbox. The mailbox holds a number of messages, so that the instance variables of the class which implements the mailbox (called MailBox) can be considered to consist of an integer to maintain a count of the number of messages, and the messages themselves. If the assumption is made that the messages are maintained by an existing *Arjuna* class called Message (which in addition to the message text records information such as the time and date of the message), then the refinement of the save_state operation the implementor of the MailBox class must provide to ensure the mailbox is recoverable and persistent would be coded in the manner illustrated in Figure 4. This figure illustrates how the primitive

```
void MailBox::save_state(ObjectState *newstate)
{
    newstate->pack(no_messages);              // save the integer

    for (int i = 0; i < no_messages; i++)
        message[i]->save_state(newstate);     // now the messages
}
```

Figure 4: The save_state operation

type (the integer) is saved in the ObjectState instance passed as an argument using the ObjectState pack operation. Since Message objects already provide a recoverable interface, the save_state operation may be invoked directly to save the state maintained by each Message object. The restore_state operation can be coded in an identical manner by simply replacing the pack with an unpack invocation, and the save_state with restore_state. Using a class such as MailBox within the scope of atomic actions (created by instantiating the class AtomicAction and invoking the operations this class provides) will ensure that *the messages maintained by the MailBox instance are persistent, will not be*

corrupted (assuming bug-free code), and any operations on the MailBox (such as deleting a message) can be recovered.

StateManager only provides a recoverable interface and therefore does not itself provide full support for persistence (or recoverability). Additional support is provided by a number of classes that manage the ObjectState instances created by (and for) the save_state (or restore_state) operation. To manage recovery *Arjuna* employs a class called RecoveryRecord. Instances of this class are created and added to the RecordList maintained by an atomic action the first time an *Arjuna* object is modified within the scope of an atomic action. Each ObjectState instance for a recoverable object is maintained in the volatile storage associated with the application program's server. Persistence is managed by a class that is refinement of RecoveryRecord called PersistenceRecord. The additional functionality provided by PersistenceRecord is the ability to save instances of ObjectState in an object store when a top-level atomic action commits (using an operation provided by StateManager called deactivate).

The classes described above are sufficient to ensure an object is recoverable, and may be made persistent. Clearly, support must also be provided for enabling a user to activate an existing persistent object. StateManager provides this capability through an operation that corresponds to the deactivate operation called activate. The activate operation is not normally invoked by the implementor or user of a class, as the class that provides support for concurrency control (LockManager) automatically invokes activate when the first lock is granted on an object (as discussed in section 2). Figure 5 shows the

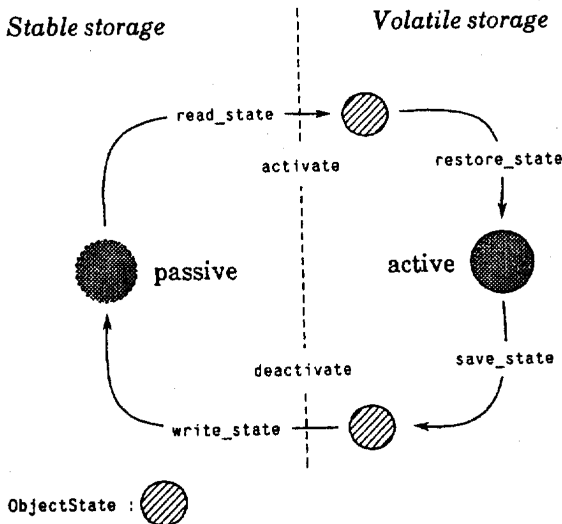

Figure 5: Object state transitions

lifetime and state transitions of a persistent object along with the operations that produce the transitions.

In addition to maintaining the state of an object contiguously, the class ObjectState ensures that the primitive types that make up the instance variables of an object are stored in a form that may be transmitted between nodes with different architectures. As a result, ObjectState instances may be sent in messages to other nodes, as required for object caching providing that a server containing the code for the object's methods is also available.

Figure 5 also illustrates how the stable storage is managed by a class called ObjectStore that provides two operations for reading (read_state) and writing (write_state) instances of ObjectState. The next section describes the design and implementation of the object store currently employed by *Arjuna*.

3.3 Object Store Organisation

To activate an existing persistent object, the passive state of the object must be located in stable storage. Given this passive state (which is an instance of ObjectState), the activate operation (executed in the server process) simply employs the restore_state operation provided by StateManager to obtain the last active state of the object. To manage and locate stable ObjectState instances, *Arjuna* employs an object store (called *Kubera*) which is implemented by the class ObjectStore. This section describes the design and implementation of this class.

The organisation of the ObjectStates in stable storage is the responsibility of the ObjectStore class. There are many approaches to organising the states, for instance, all objects could be maintained in a contiguous portion of stable storage, or alternatively, objects could be clustered together to improve the speed with which a set of common objects are activated. The approach adopted by the ObjectStore class is a mixture of these approaches.

Since the identity of an object is a function of the class of the object, the name of the class may be used to structure the object store organisation. In this way, objects which are instances of the same class can be maintained together, thereby partitioning the name space of the object store and improving the speed taken to locate the passive state of an object. Furthermore, the entire (logical) object store could be implemented as a collection of individual object stores, one for each class of objects, each containing all the (passive states of) instances of that class. If each object store is itself a persistent object, then all objects and classes will be reachable using the object store that contains the passive states of the object store objects.

Using the classes described in this paper to implement this organisation, the class ObjectStore would be responsible for maintaining the physical location of all ObjectState instances in stable storage. Since each ObjectStore instance is a persistent object (the class ObjectStore would be derived from LockManager),

the passive state of these instances would be maintained by another ObjectStore instance which could be considered to be the *root* object store. The passive state of each ObjectStore would therefore be the location in stable storage of the ObjectState objects held by that ObjectStore instance.

To locate the passive state for an object, the object store for the class of that object would be located using the root object store. From the class's object store the state of the desired instance could then be simply found. For example, consider how the passive state of an instance of the MailBox class called Arjuna would be held in stable storage, and relationship between the ObjectStore objects needed to access this instance. Figure 6 illustrates both the logical and

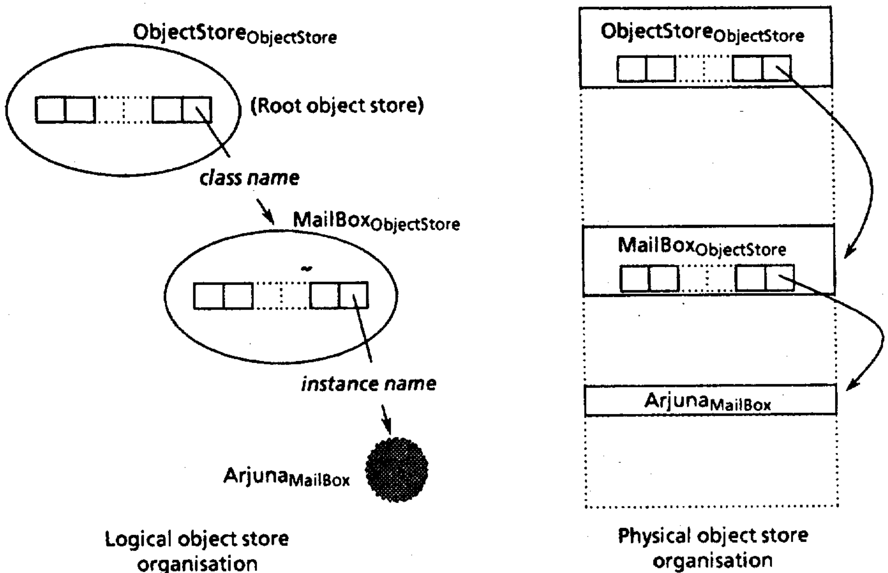

Figure 6: Logical and physical object store organisations

physical relationship between the MailBox and ObjectStore objects.

Maintaining all instances of a class together in an object store has advantages beyond those of simplifying the management of the objects in stable storage. For instance, if it is common when accessing a persistent object to find the object unavailable (as another user may be accessing the object in a manner which would conflict with other users), then the class could take advantage of the object store organisation. An implementation of the class could be made so that another instance of the class would be used in place of the original simply by iterating over the contents of the object store for other instances of that class. Another advantage, given the object store organisation described above, is that *a user can easily create their own object stores (the passive state of each being maintained in the root object store) to manage any classes they may define.*

A very simple implementation of this design has been made and is currently used by persistent objects in *Arjuna*. This implementation does not employ stable storage but instead uses the file system of the underlying operating system (Unix). Each object store is implemented using a directory, so that the entries in the directory are the instances of the class that have been created. Each object is kept in a file so that the entry in the directory is the name of the file. This name is the unique identifier (uid) of the object, with the name of the directory being the class name. A single directory at a fixed location on each node represents the root object store, with this directory containing the directories for all classes created on that node.

Clearly, an implementation using the Unix file system has performance penalties, but it has enabled the abstractions developed to be quickly implemented and easily tested. A new implementation is currently in progress that implements the previously developed interface (of the ObjectStore class), but which is optimised for the storage and retrieval of passive states of persistent objects.

4 A SIMPLE EXAMPLE

The previous sections of the paper have described the principles and underlying structure of *Arjuna* for managing persistent objects. This section describes an example intended to illustrate some of those principles in practice. The example chosen is that of a spreadsheet. For simplicity it is assumed that the spreadsheet is represented by a two-dimensional integer matrix with a fixed set of dependencies that state that the last entry in each row (and column) contains the sum of the preceding elements of the row (column). One possible C++ class definition for this simple style of spreadsheet is given below as Figure 7.

```
class SpreadSheet : public LockManager
{
    int Elements[SPRDSHT_SIZE][SPRDSHT_SIZE];

public:
    SpreadSheet(int*, ArjunaName* =0);
    ~SpreadSheet( );

    Outcome Set(int, int, int);
    Outcome Get(int, int, int*);

    // Arjuna specific operations

    virtual void save_state(ObjectState*);
    virtual void restore_state(ObjectState*);
    ...
};
```

Figure 7: The SpreadSheet class

This definition shows the additional measures that need to be taken to use SpreadSheet *type objects within the Arjuna system. Firstly, the class is derived*

from the *Arjuna* provided class LockManager. This makes the concurrency control, recovery and persistence capabilities of *Arjuna* available to the implementor of the operations of the SpreadSheet class. Secondly, several *Arjuna* specific operations are added to the basic set required by the normal implementation of the class. This latter set of operations provides type-specific operations used by *Arjuna* when saving and restoring instances of this class to and from the object store. In addition the constructor for the SpreadSheet class takes an instance of the class ArjunaName as a parameter so that the appropriate instance will be automatically activated from the object store when necessary as was described in the previous section.

Given this simple class definition one possible implementation of the Set operation is illustrated as Figure 8. This operation sets one element of the

```
// Set(x, y, v) changes elements x, y of spreadsheet to have value v.
// If successful returns SUCCESS else type of failure which occurred.
// Performed as an atomic action.

Outcome SpreadSheet::Set(int x, int y, int v)
{
    // Check parameters are ok

    if ((x < 0) || (y < 0) || (x >= SPRDSHT_SIZE) || (y >= SPRDSHT_SIZE))
        return ILLEGAL_ARGS;

    // Parameters are ok - now do the real work

    AtomicAction SetAction;
    Outcome result = SUCCESS;

    SetAction.Begin( );                          // Begin action

    if (setlock(new Lock(WRITE)) == GRANTED)     // and try for lock
    {                                            // locked - update
        int OldValue = Elements[x][y];           // Save old value
        Elements[x][y] = v;                      // Change value
        Elements[SPRDSHT_SIZE][y] += v - OldValue;
        Elements[x][SPRDSHT_SIZE] += v - OldValue;
        Elements[SPRDSHT_SIZE][SPRDSHT_SIZE] += v - OldValue;

        if (SetAction.End( ) != COMMITTED)       // Try to commit action
            result = COMMIT_FAILED;
    }
    else                                         // Lock refused
    {
        if (SetAction.Abort( ) != ABORTED)       // So abort action
            result = ABORT_FAILED;
    }
    return result;
}
```

Figure 8: The Set operation of SpreadSheet

spreadsheet to a new value and updates the totals as appropriate. The operation

is performed as an atomic action so that it either succeeds or fails in its entirety. Once the parameters to the operation have been verified an atomic action is started, and an attempt is made to lock the object in WRITE (exclusive) mode. If this succeeds then the object will have been activated and loaded from the object store automatically (if necessary) by the *Arjuna* system, before the last active state held by the object is updated.

After the update, an attempt is made to commit the atomic action and if this is successful then the operation returns with a value indicating that it has succeeded, otherwise the atomic action is aborted and the value returned indicates that the operation has failed. In either case no further interaction with the *Arjuna* system is required since the system will ensure that the modified object state is returned to the object store (providing the atomic action committed) and any locks that were set are automatically released.

Activation of the object (loading from the object store) takes place automatically at the appropriate moment and makes use of the implementor-provided operation restore_state. An implementation of this operation is illustrated in Figure 9, showing how the primitive facilities provided by *Arjuna* can be used to

```
void SpreadSheet::restore_state(ObjectState *S)
{
    for (int x = 0; x <= SPRDSHT_SIZE; x++)
        for (int y = 0; y <= SPRDSHT_SIZE; y++)
            S->unpack(&Elements[x][y]);
}
```

Figure 9: restore_state operation for SpreadSheet

unpack the individual elements of the spreadsheet from the supplied ObjectState instance.

This example, albeit very simple, has shown the fundamentals of programming in the *Arjuna* system. At its simplest, this amounts to little more than deriving the user-defined class from LockManager providing appropriate definitions for the *Arjuna* operations save_state, restore_state, etc; setting appropriate locks to govern the level of concurrency the class supports; and declaring and using atomic actions to control the recovery requirements of the class and any application programs.

5 CONCLUDING REMARKS

This section summarises the research work presented here and puts it in context of other published work in this area. In the first section it was stated that object management mechanisms for distribution, persistence, recovery, concurrency control etc. should be both integrated and flexible. The computational model of atomic actions controlling operations on objects provides the framework for developing these mechanisms in an integrated manner. These mechanisms have been incorporated in *Arjuna* by designing and implementing a number of classes related by the inheritance hierarchy

shown in Figure 2. This use of type inheritance makes the *Arjuna* recovery and concurrency mechanisms very flexible, allowing default behaviour to be automatically derived yet permitting type-specific modifications (see [Dixon and Shrivastava 87], and [Parrington and Shrivastava 88] for examples). This approach is also adaptable since no new language features need be introduced. Clearly there are potential penalties since careless programming such as omitting to invoke the locking operations, can create chaos. For this reason some research projects, such as Emerald [Black *et al.* 87] and Avance [Bjornerstedt and Britts 88], have taken the step of designing new languages for distributed object-oriented programming. Our implementation approach can be applied readily to such special language-based systems.

All current research efforts on incorporating fault-tolerance and persistence in object systems make use of atomic actions (atomic transactions) for manipulating persistent objects. However two differing approaches have emerged regarding the use of atomic actions. In one approach, characterised by the system described here, atomic actions are made application level entities, to be used in application specific manner; whilst this is not the case in the second approach where actions are utilised by the underlying system for making object states persistent on the object store. Some examples of systems broadly following this approach are Jasmine [Wiebe 86], Gemstone [Maier *et al.* 86]. This second approach is convenient for integrating an object based programming system into a database system supporting atomic transactions. Notwithstanding these differences, some core mechanisms for object manipulation are required in all systems. For example, the Jasmine mechanisms for activating, deactivating and packing objects for network transmission are broadly similar to ours. The persistent object store system for Smalltalk-80 reported in [Low 88] provides application level atomic actions similar to ours, but its implementation for atomic action support is quite different. Rather than providing a common support for recovery, concurrency and persistence (*Arjuna* class StateManager), a new Transaction Manager subsystem has been developed, independent of the object store. A comparative evaluation of these systems - including ours - can only be made when application development experience becomes available.

The Avalon/C++ [Herlihy and Wing 87] system comes closest to *Arjuna* in its use of the type inheritance approach for implementing atomic actions. However, Avalon/C++ is different because it provides an object-oriented interface to an existing transaction processing system (Camelot), whereas *Arjuna* has to implement most of the functionality provided by Camelot as well. Both Avalon/C++ and *Arjuna* are good examples of working systems illustrating that type inheritance provides an effective way to construct flexible fault-tolerant and distributed systems.

As was reported in section 2.3, major parts of Arjuna have been designed and implemented. Future work will include the design of a name service and the implementation of object caching mechanisms. The current implementation of

Kubera exploits the simple functionality of the underlying filing system; enhancing the object store to include features found in object based database systems such as ORION [Garza and Kim 88] represents a worthwhile extension. The existing implementation can be adapted easily for the caching of *immutable objects*. However, the current design of LockManager will need some enhancements before other types of objects can be cached without compromising consistency.

ACKNOWLEDGEMENT

The work reported here has been supported in part by a grant from the UK Science and Engineering Research Council (under the Alvey Initiative in Software Engineering) and by a contract from Architecture Projects Management Ltd. of Cambridge (UK).

REFERENCES

Atkinson *et al.* 83

Atkinson, M.P., P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison, "An Approach to Persistent Programming," The Computer Journal, Vol. 26, No. 4, pp. 360-365, 1983.

Bernstein *et al.* 87

Bernstein, P.A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

Bjornerstedt and Britts 88

Bjornerstedt , A., and S. Britts, "AVANCE: An Object Management System," *OOPSLA '88 Conference Proceedings*, pp. 206-221, September 1988.

Black *et al.* 87

Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 65-76, January 1987.

Cox 86

Cox, B.J., *Object Oriented Programming*, Addison Wesley, 1986.

Dasgupta *et al.* 85

Dasgupta, P., R.J. LeBlanc Jr., and E. Spafford, "The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System," Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.

Dixon and Shrivastava 87

Dixon, G.N., and Shrivastava, S.K., "Exploiting Type-Inheritance Facilities to Implement Recoverability in Object Based Systems",

Proceedings of 6th Symposium on Reliability in Distributed Software and Database Systems, Williamsburg, pp. 107-114, March 1987.

Dixon 88
Dixon, G.N., "Object Management for Persistence and Recoverability," Ph.D Thesis, Technical Report TR276, Computing Laboratory, University of Newcastle upon Tyne, July 1988.

Eswaran *et al.* 76
Eswaran, K.P, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976.

Garza and Kim 88
Garza, J. and W. Kim, "Transaction Management in an Object-Oriented Database System," *Proceedings of the SIGMOD Conference on Management of Data*, Chicago, Illinois, pp.37-45, September 1988.

Gray 78
Gray, J.N., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, eds. R. Bayer, R.M. Graham, and G. Seegmueller, pp. 393-481, Springer, 1978.

Herbert and Monk 87
Herbert, A.J., and J.Monk, (eds) "ANSA reference manual," 1987.

Herlihy and Wing 87
Herlihy, M.P., and J.M. Wing, "Avalon: Language Support for Reliable Distributed Systems," *Digest of Papers, FTCS-17, Seventeenth Annual Symposium on Fault-Tolerant Computing*, Pittsburgh, pp. 89-94, July 1987.

Liskov 88
Liskov, B. "Distributed Programming in Argus," *Communications of the ACM*, Vol. 31, No. 3, pp. 300-312, March 1988.

Low 88
Low, C, "A Shared Persistent Object Store," *Proceedings of the Second European Conference on Object-Oriented Programming*, ECOOP88, pp. 390-408, Oslo, Norway, August 1988.

Maier *et al.* 86
Maier, D., J. Stein, A. Otis, and A. Purdy, "Development of a Object-Oriented DBMS," *OOPSLA '86 Conference Proceedings*, pp. 472-482, September 1986.

Oppen and Dalal 81
Oppen, D.C., and Y.K. Dalal, "The Clearinghouse: A Decentralized Agent

for Locating Named Objects in a Distributed Environment," Xerox Office Products Division, Palo Alto, 1981.

Panzieri and Shrivastava 88
Panzieri, F., and S.K. Shrivastava, "Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing," *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 1, pp. 30-37, January 1988.

Parrington 88
Parrington, G.D., "Management of Concurrency in a Reliable Object-Oriented System," Ph.D Thesis, Technical Report TR277, Computing Laboratory, University of Newcastle upon Tyne, July 1988.

Parrington and Shrivastava 88
Parrington, G.D., and S.K. Shrivastava, "Implementing Concurrency Control for Robust Object-Oriented Systems," *Proceedings of the Second European Conference on Object-Oriented Programming*, ECOOP88, pp. 233-249, Oslo, Norway, August 1988.

Shrivastava *et al.* 89
Shrivastava, S.K., G.N. Dixon, G.D. Parrington, F. Hedayati, S.M. Wheater, and M.C.Little "The Design and Implementation of *Arjuna*," Technical Report TR280, Computing Laboratory, University of Newcastle upon Tyne, March 1989.

Stroustrup 86
Stroustrup, B., *The C + + Programming Language*, Addison Wesley, 1986.

Wiebe 86
Wiebe, D., "A Distributed Repository for Immutable Persistent Objects," *OOPSLA '86 Conference Proceedings*, pp. 453-465, September 1986.