The Design and Implementation of Arjuna

Graham D Parrington, Santosh K Shrivastava, Stuart M Wheater and Mark C Little

Department of Computing Science
The University of Newcastle upon Tyne
Newcastle upon Tyne
NE1 7RU UK

Arjuna is an object-oriented programming system, implemented entirely in C++, that provides a set of tools for the construction of fault-tolerant distributed applications. *Arjuna* exploits features found in most object-oriented languages (such as inheritance) and only requires a limited set of system capabilities commonly found in conventional operating systems. *Arjuna* provides the programmer with classes that implement atomic transactions, object level recovery, concurrency control and persistence. These facilities can be overridden by the programmer as the needs of the application dictate. Distribution of an *Arjuna* application is handled using stub generation techniques that operate on the original C++ class headers normally used by the standard compiler. The system is portable, modular and flexible. The paper presents the design and implementation details of *Arjuna* and takes a retrospective look at the system based on the application building experience of users.

1. Introduction

Arjuna is an object-oriented programming system, implemented in C++, that provides a set of tools for the construction of fault-tolerant distributed applications. *Arjuna* supports the computational model of *nested atomic actions* (nested atomic transactions) controlling operations on persistent (long-lived) objects. *Arjuna* objects can be replicated on distinct nodes in order to obtain high availability.

Several other prototype distributed object-oriented systems have been built, often emphasising different aspects of fault tolerance, distribution, concurrency, persistence etc. Their implementations have been achieved by the creation of new programming languages, enhancements to existing languages and systems, creation of entirely new operating systems or by some combination of these (for example: *Emerald* [Black87], *Clouds* [Dasgupta91], *Avalon* [Detlefs88], *Argus* [Liskov88], *SOS* [Shapiro89], *Guide* [Balter91], and *Choices* [Campbell93]). We wanted *Arjuna* to be a widely available toolset, in the spirit of other UNIX related software (e.g., the X window system). Thus we have deliberately chosen not to modify the language or the operating system. The *ISIS* system [Birman87, Birman93] is one of the best known examples of a software system that has also taken the same approach. However, *ISIS* is concerned with the provision of reliable group

communications infrastructure, while *Arjuna* deals with higher level application building tools. A careful examination of the underlying models of these systems (virtual synchrony and serialisable transactions respectively) and their integration is an interesting topic for further research.

Designing and implementing a programming system capable of supporting such 'objects and actions' based applications by utilising *existing* programming languages and distributed system services is a challenging task. Firstly, despite their obvious potential, most widely used object-oriented languages have little or no direct support for programming a distributed application. The primary reason for this focus is that existing object-oriented languages have been developed without the demands of distribution in mind and thus possess one or more features that are either impossible, or at least impractical, to support in a distributed execution environment. A classic example of such a feature being the assumption that the application will execute within a single address space. Secondly, support for distributed computing on currently available systems varies from the provision of bare essential services, in the form of networking support for message passing, to slightly more advanced services for interprocess communication (e.g., remote procedure calls), naming and binding (for locating named services) and remote file access. The challenge lies in integrating these services into an advanced programming environment permitting existing languages to be used for building distributed applications.

The design and implementation goal of *Arjuna* 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:

(i) *Modularity.* The system should be easy to install and run on a variety of hardware and software configurations. In particular, it should be possible to replace a component of *Arjuna* by an equivalent component already present in the underlying system.

(ii) *Integration of mechanisms.* A fault-tolerant distributed system requires a variety of system functions for naming, locating and invoking operations upon local and remote objects, and 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.

(iii) *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.

In *Arjuna,* 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. This facilitates the task of implementing the architecture on a variety of systems with differing support for distributed computing [Shrivastava94a]. For example, it is relatively easy to replace the default RPC module of *Arjuna* by a different one. The remaining two goals have been met primarily through the provision of a C++ class library; the classes in this library have been organised into a class hierarchy in a manner that will be familiar to the developers of

more traditional (single node) centralised object-oriented systems. *Arjuna* assumes that every major entity in the application is an object. This philosophy also applies to the internal structure of *Arjuna* itself. Thus, *Arjuna* 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 and distribution in a very flexible and integrated manner.

The *Arjuna* research effort began in late 1985. Early papers [Dixon87, Parrington88, Dixon89] describe preliminary versions of the recovery, concurrency control and persistence mechanisms of *Arjuna*. The use of C++ in building reliable distributed applications is examined in [Parrington90, Parrington95], while a brief overview of the system appears in [Shrivastava91]. This is the first paper that describes the system as a whole, giving the overall architecture of *Arjuna*, including its design, implementation and performance details of the major system components responsible for maintaining the abstraction of distribution transparency. We will concentrate on the core features of *Arjuna* that are concerned with the provision of atomic actions and persistent objects, and only briefly discuss more advance features concerned with the provision of object replication and clustering (details concerning these may be found in [Little93, Little94 and Wheater94]). All aspects of the system described here have been fully tested and implemented to run on networked UNIX systems. *Arjuna* has been used for building a number of applications. Based on this experience, in the last section of this paper we will examine how effectively we have met the stated goals Finally, *Arjuna* system software has been freely available for research, development and teaching purposes since 1992 (for information consult our WWW page: http://arjuna.ncl.ac.uk/).

2. Failure Assumptions and Computation Model

2.1. Failure assumptions

It will be 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. More serious are network partition failures preventing functioning nodes from communicating with each other. Because not every communication environment is subject to partitions, we categorise communication environments into two types:

(i) *Non-partitionable networks*. In such a network, functioning nodes are capable of communicating with each other, and judiciously chosen time-outs together with network level ‘ping’ mechanisms can act as an accurate indication of node failures.

(ii) *Partitionable networks*. Here physical breakdowns (e.g., a crash of a gateway node) and/or network congestion can prevent communication between functioning nodes. In such networks, time-outs and network level 'ping' mechanisms cannot act as an accurate indication of node failures (they can only be used for *suspecting* failures). We assume that a partition in a partitionable network is eventually repaired.

2.2. Objects and Actions

As indicated, we are considering a computation model in which application programs manipulate persistent (long-lived) objects under the control of atomic actions (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. We will consider an application program initiated on a node to be the *root* of a computation. Distributed execution is achieved by invoking operations on objects which may be remote from the invoker. An operation invocation upon a remote object is performed via a remote procedure call (RPC). All operation invocations may be controlled by the use of atomic actions which have the well known properties of *serialisability*, *failure atomicity*, and *permanence of effect.* Atomic actions can be nested. A commit protocol is used during the termination of an outermost atomic action (*top-level action*) to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or, if the atomic action 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 actions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

The object and atomic action 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 server for receiving RPC invocations.

In the model discussed above, a persistent object can become *unavailable* due to failures such as a crash of the object server, or network partition preventing communications between clients and the server. The *availability* of an object can be increased by replicating it on several nodes. We will consider the case of *strong consistency* which requires that all replicas that are regarded as *available* be mutually consistent (so the persistent states of all available replicas are required to be identical). Object replicas must therefore be managed through appropriate replica-consistency protocols to ensure strong consistency. To tolerate K replica failures, in a non-partitionable network, it is necessary to maintain at least K+1 replicas of an object, whereas

in a partitionable network, a minimum of 2K+1 replicas are necessary to maintain availability in the partition with access to the majority of the replicas (the object becomes unavailable in all of the other partitions).

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 for each persistent object there is one node (say α) which, if functioning, is capable of running an *object server* which can execute the operations of that object (in effect, this would require that α has access to the executable binary of the code for the object's methods as well as the persistent state of the object stored on some, possibly remote, object store). Before a client can invoke an operation on an object, it must first be *connected* or *bound* to the object server managing that object. It will be the responsibility of a node, such as α, to provide such a connection service to clients. If the object in question is in a passive state, then α is also responsible for activating the object before connecting the requesting client to the server. In order to get a connection, an application program must be able to obtain *location* information about the object (such as the name of the node where the server for the object can be made available). We assume that each persistent object possesses a unique, system given identifier (UID). In our model an application program obtains the location information in two stages:

(i) By first presenting the application level name of the object (a string) to a globally accessible *naming service*; assuming the object has been registered with the naming service, the naming service maps this string to the UID of the object.

(ii) The application program then presents the UID of the object to a globally accessible *binding service* to obtain the location information. Once an application program (client) has obtained the location information about an object it can request the relevant node to establish a connection (binding) to the server managing that object. The typical structure of an application level program is shown below:

<create bindings>;
<invoke operations from within atomic actions>;
<break bindings>

In our model, bindings are not stable (they do not survive the real or suspected crash of the client or server). Bindings to servers are created as objects enter scope in the application program. If some bound server subsequently crashes (or gets disconnected) then the corresponding binding is broken and not repaired within the lifetime of the program (even if the server node is functioning again); all the surviving bindings are explicitly broken as objects go out of the scope of the application program.

The passive representation of an object in the object store may differ from its volatile store representation (e.g., pointers may be represented as offsets or UIDs). Our 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. The server of an activated object can then use these operations during abort or commit processing. Further, we assume that each object is responsible for performing appropriate concurrency control to ensure serialisability of atomic actions. 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 *Arjuna* and the services they provide for supporting persistent objects.

(i) *Atomic Action module.* Provides atomic action support to application programs in the form of operations for starting, committing and aborting atomic actions;

(ii) *RPC module.* Provides facilities to clients for connecting (disconnecting) to object servers and invoking operations on objects;

(iii) *Naming and Binding module.* Provides a mapping from user-given names of objects to UIDs, and a mapping from UIDs to location information such as the identity of the host where the server for the object can be made available;

(iv) *Object Store module.* Provides a stable storage repository for objects; these objects are assigned unique identifiers (UIDs) for naming them.

The relationship amongst these modules is depicted in Figure 1. Every node in the system will provide the RPC and Atomic Action modules. Any node capable of providing stable object storage will in addition contain an Object Store module. Nodes without stable storage may access these services via their local RPC module. The Naming and Binding module is not necessary on every node since its services can also be utilised through the services provided by the RPC module. This system structure is highly modular: by encapsulating the properties of persistence, recoverability, shareability, serialisability and failure atomicity in an Atomic Action module and defining narrow, well-defined interfaces to the supporting environment, we achieve a significant degree of modularity as well as portability for *Arjuna* [Shrivastava94a].

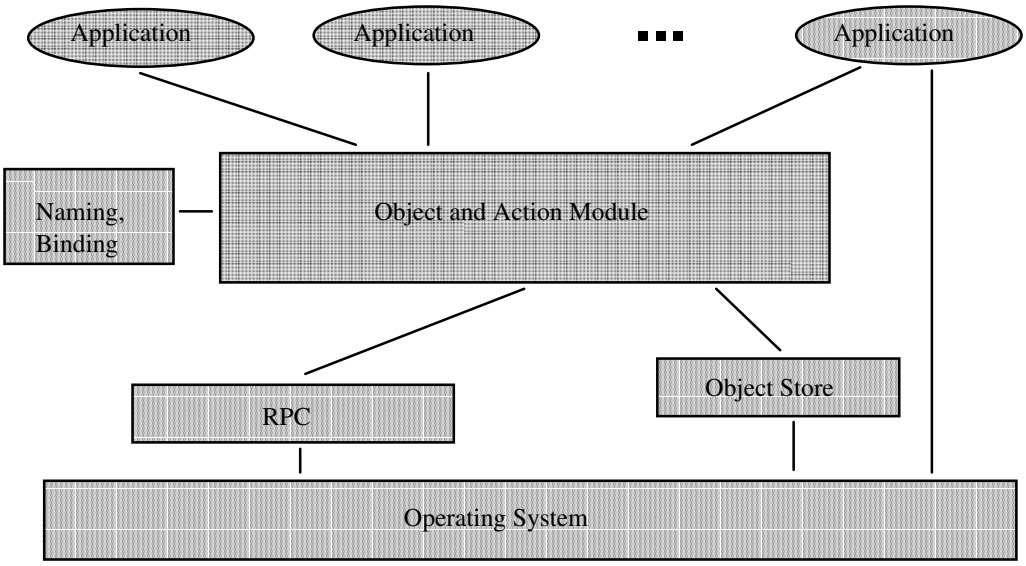

Figure 1: Components of *Arjuna*

In *Arjuna* the primitive operation `initiate(...)`, provided by the RPC module is used for binding to an object. A complementary operation, called `terminate(..)`, is available for breaking a binding. Clients and servers have communication identifiers, CIDs (UNIX sockets in the current implementation), for sending and receiving messages. The RPC module of each node has a connection manager process that is responsible for creating and terminating bindings to local servers. The *Arjuna* stub generation system for C++ generates the necessary client-server stub-codes for accessing remote objects via RPCs and also generates calls on `initiate` and `terminate` as an object comes and goes out of the scope of a computation [Parrington90, Parrington95].

An RPC has the following semantics: a normal termination will indicate that a reply message containing the results of the execution has been received from the server; an exceptional return will indicate that no such message was received, and the operation may or may not have been executed. Once the execution of an action begins, any failures preventing forward progress of the computation lead to the action being aborted, and any updates to objects undone. However, as establishing and breaking bindings can be performed outside of the control of any application level atomic actions, it is instructive to enquire how any clean-up is performed if client, server or partition failures occur before (after) an application level action has started (finished). The simple case is the crash of a server node: this has the automatic effect of breaking the connection with all of its clients; if a client subsequently enters an atomic action and invokes an operation in the server, the invocation will return exceptionally and the action will be aborted; on the other hand, if the client is in the process of breaking the bindings then this has occurred already. More difficult is the case of a client crash. Suppose the client crashes after binding to a server. Then explicit steps must be taken to remove any state information kept for the *orphaned* bindings; this requires that a server node must have a mechanism for breaking the binding if it suspects the crash of a client. This mechanism will also cope with a partition that prevents any communication between a client and a server. The *Arjuna* RPC level facilities for the detection

and killing of orphans [Panzieri88] are responsible for such a cleanup, ensuring at the same time that an orphaned server (a server with no bindings) is terminated.

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(Name-A);          /* bind to A */
     O2 objct2(Name-B);          /* bind to B */
     AtomicAction act;
     act.Begin();                /* start of atomic action act */
     objct1.op(...);
     objct2.op(...);             /* invocations ....*/
     .......
     act.End();                  /* act commits */
 }                               /* break bindings to A and B */
```

Program 1: Outline Action Example

Thus, to bind to A, a local instance of O1 called objct1 is created, passing to its constructor an instance of the *Arjuna* naming class ArjunaName (which will be described in more detail in section 4.5) called Name-A suitably initialised with information about A (e.g., its UID, the location of the server node, etc.). This enables the client side stub-constructor to *initiate* A, resulting in binding to the server for A. The Object Store module of *Arjuna* enables a server 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.

Now assume that the client program is executing at node N_1 and the server node for A is at N_2 (see Figure 2). The client process at node N_1 executing the stub for object1 is responsible for invoking the initiate operation of the local RPC module in order to send a connection request to the connection manager at N_2. The connection manager locates the object sever for A who then returns the CID to the client at N_1, thereby terminating the invocation of initiate at N_1. The storage and retrieval of object states from an object store is managed by a *store daemon*. The object server uses the store demon for retrieving the state of an object from the object store. For efficiency reasons, an object server can (and will) directly access the object store, bypassing the daemon if the server and the store are on the same node. However, if the object store is remote, then it must contact the store demon of the remote node managing the object store.

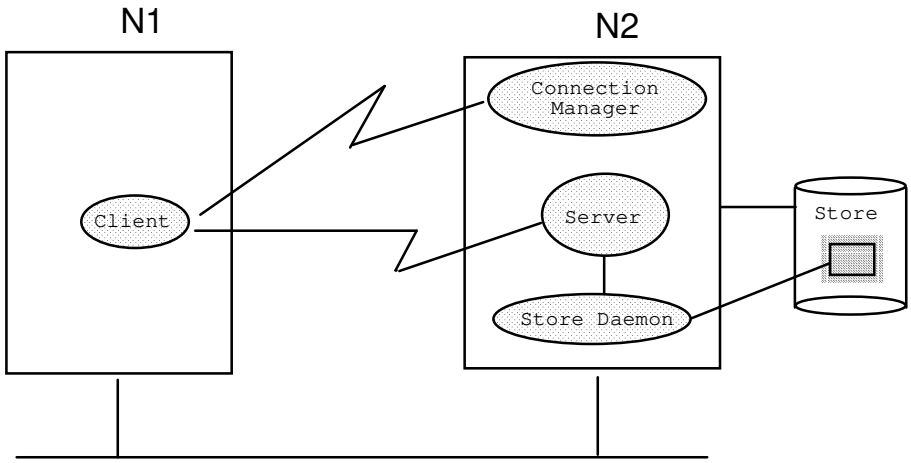

Figure 2: Accessing an Object

To manipulate objects under the control of an atomic action, the client creates a local instance of an action (`act`) and invokes its `Begin` operation. The `End` operation is responsible for committing the atomic action (using a two-phase commit protocol). When an object goes out of scope, it is destroyed by executing its destructor. As a part of this, the client-side destructor (e.g., the stub destructor for `object1`) breaks the binding with the object server at the remote node (using the operation `terminate`).

4. Implementing the System

The following sub-sections describe in detail how the architecture outlined in the previous sections has been implemented in *Arjuna*. The actual implementation effectively splits into two distinct areas. Firstly, there is a set of C++ classes (effectively organised as a single inheritance hierarchy) that implement a non-distributed version of the system. Secondly, there is a stub generation and RPC system to handle the distribution aspects of the system. We start by describing the first part of the system, concentrating on object storage, retrieval, and the atomic action system.

Although deliberately not machine or system specific *Arjuna* still requires certain basic capabilities from the underlying operating system, these mainly include:

(i) BSD style sockets; these are needed by the supplied RPC mechanism.

(ii) System V shared memory and semaphore support is required by both the RPC mechanism and the concurrency controller.

(iii) Support for long file names which are generated by the object persistence mechanisms.

4.1. The Life Cycle of an Arjuna 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 *Arjuna* 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 `ObjectState` 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 C++ object to be converted to and from an `ObjectState` instance. The fundamental life cycle of a persistent object in *Arjuna* is shown in Figure 3.

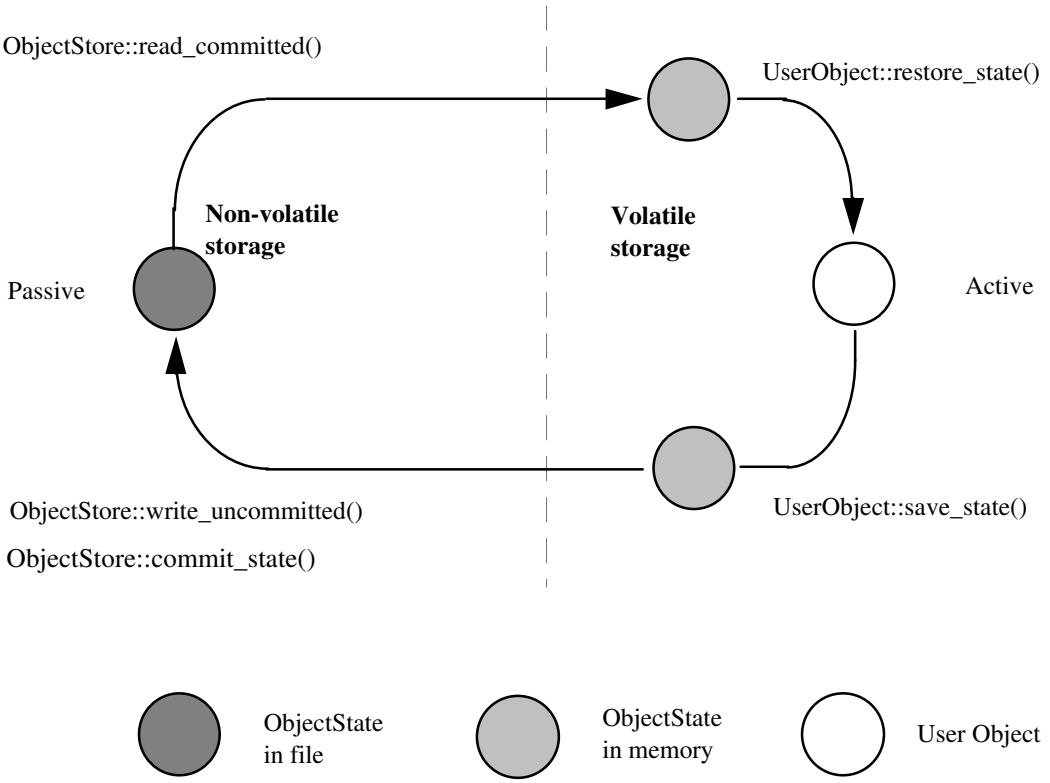

Figure 3: The Life Cycle Of A Persistent Object

(i) The object is initially passive, and is stored in the object store as an instance of the class `ObjectState`.

(ii) 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 `ObjectState` instance into a fully-fledged object by the `restore_state` operation of the object.

(iii) When the application has finished with the object it is *deactivated* by converting it back into an `ObjectState` 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

Arjuna 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 `ObjectState` and `Buffer`.

The `Buffer` class maintains an internal array into which instances of the standard types can be contiguously packed (unpacked) using the overloaded `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. Any other architecture independent format (such as XDR or ASN.1) could be implemented simply by replacing the operations of `Buffer` with ones appropriate to the encoding required.

```
class Buffer
{
public:
     Buffer (size_t buffSize = DEFAULT_CHUNK_SIZE);
     virtual ~Buffer ();

     char *buffer () const;
     size_t length () const;

     /* pack and unpack operations for standard C++ types */

     Boolean pack (char);
     Boolean pack (unsigned char);
     Boolean pack (int);
     Boolean pack (unsigned int);
     /* and so on for other standard types */
     . . .

     Boolean unpack (char&);
     Boolean unpack (unsigned char&);
     Boolean unpack (int&);
     Boolean unpack (unsigned int&);
     . . .
private:
     char *bufferStart;
     . . .
};
```

```
class ObjectState : public Buffer
{
public:
     ObjectState (const Uid& newUid, TypeName tName);
     ~ObjectState ();

     Boolean notempty () const;
     size_t size () const;
     const Uid& stateUid () const;
     const TypeName type () const;

private:
     Uid buffer Uid;
     TypeName imageType;
};
```

Program 2: classes Buffer and ObjectState

The class `ObjectState` provides all the functionality of `Buffer` (through inheritance) but adds two additional instance variables that signify the `UID` and `Type` of the object for which an `ObjectState` instance is a compacted 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 *Arjuna* deliberately has a fairly restricted interface so that it can be implemented in a variety of ways. For example, the currently available distribution has object stores implemented in shared memory; on the UNIX file system (in four different forms); and as a remotely accessible store (implemented using the stub generation techniques described in a later section). This is implemented by making the `ObjectStore` class *abstract* and deriving the classes that handle the actual implementation of each style of object store from this base class.

The object store only stores and retrieves instances of the class `ObjectState`. 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 `write_(un)committed` operations. 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 *Arjuna* system components as appropriate thus the existence of any shadow versions of objects in the store are hidden from the programmer.

```
class ObjectStore
{
public:
     enum StateType { OS_SHADOW, OS_ORIGINAL, OS_INVISIBLE };
     enum StateStatus { OS_UNKNOWN = 0, OS_COMMITTED = 1, OS_UNCOMMITTED = 2,
                        OS_HIDDEN = 4,
                        OS_COMMITTED_HIDDEN = OS_COMMITTED | OS_HIDDEN,
                        OS_UNCOMMITTED_HIDDEN = OS_ UNCOMMITTED | OS_HIDDEN };

     virtual ~ObjectStore ();

     /* The abstract interface */
     virtual StateStatus currentState (const Uid&, const TypeName) = 0;
     virtual Boolean commit_state (const Uid&, const TypeName) = 0;
     virtual ObjectState *read_committed (const Uid&, const TypeName) = 0;
     virtual ObjectState *read_uncommitted (const Uid&, const TypeName) = 0;
     virtual Boolean remove_committed (const Uid&, const TypeName) = 0;
     virtual Boolean remove_uncommitted (const Uid&, const TypeName) = 0;
     virtual Boolean write_committed (const Uid&, const TypeName, const ObjectState&) = 0;
     virtual Boolean write_uncommitted (const Uid&, const TypeName, const ObjectState&) = 0;

     virtual const TypeName type () const = 0;
     virtual void storeLocation (const char*) = 0;

     ...
     static ObjectStore* create (const TypeName, const char*);
     static void destroy (ObjectStore*&);

protected:
     ObjectStore ();
     ...
};
```

Program 3: class ObjectStore

The implementations of the object store provided in the standard release map the inheritance hierarchy for an object (provided by the `type()` operation as a string such as "`StateManager/LockManager/Object`") directly onto a standard UNIX directory structure. Each individual object is then stored as an individual file named by the UID of the object. Since the object store must be capable of storing shadow copies of an object in addition to the original, in the default implementation, each file is segmented into three areas: a header, and two object storage areas. The header block contains a control structure that describes the size and offsets of the other two storage areas, together with flags that indicate which area represents the original and which the shadow copy and whether the states are currently visible. The areas all start on disk block boundaries and the header block always has the first block reserved for it. Reading or writing a state then consists of reading the header (to determine which area contains the appropriate state) and then reading (or writing) the state requested. Committing a shadow copy is achieved by simple manipulation of the header block.

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 Action Services

4.3.1. Overview

The principal classes which make up the class hierarchy of *Arjuna* Atomic Action 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 action control abstraction
     AbstractRecord         // Important utility class
          RecoveryRecord    // handles object recovery
          LockRecord        // handles object locking
          RecordList        // Intentions list
          other management record types
```

To make use of atomic actions 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`, `End`) can then be used to start and manipulate atomic actions (including nested actions). The only objects controlled by the resulting atomic actions are those objects which are either instances of *Arjuna* classes or are user-defined classes derived from `LockManager` and hence are members of the hierarchy shown above. Most *Arjuna* 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 actions. The implementation of atomic action 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 Arjuna class `LockManager`. An application containing an atomic action `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:

(i) Check write lock compatibility with the currently held locks, and if allowed:

(ii) 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`.

(iii) Create and insert a `LockRecord` instance in the `RecordList` of `Trans`.

Now suppose that action `Trans` is aborted sometime after the lock has been acquired. Then the `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 *Arjuna* 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).

```
enum ObjectStatus
{
     PASSIVE, PASSIVE_NEW, ACTIVE, ACTIVE_NEW
};

enum ObjectType
{
     RECOVERABLE, ANDPERSISTENT, NEITHER
};

class StateManager
{
public:
     Boolean activate (const char * = 0);
     Boolean deactivate (const char * = 0, Boolean commit = TRUE);

     Uid get_uid () const;

     virtual Boolean restore_state (ObjectState&, ObjectType) = 0;
     virtual Boolean save_state (ObjectState&, ObjectType) = 0;
     virtual const TypeName type () const = 0;

protected:
     /* Constructors & destructor */

     StateManager (ObjectType ot = RECOVERABLE);
     StateManager (const Uid&);
     virtual ~StateManager ();
     . . .
     void modified ();

private:
     . . .
};
```

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 `ObjectState` 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 since `StateManager` does not have access to a *runtime* description of the layout of an arbitrary C++ object in memory and thus cannot implement a default policy for converting the in memory version of the object to its passive form. If a different language that supported a runtime type identification system had been

used to implement the system this requirement could have been removed. However, the capabilities provided by `ObjectState` 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:

```
Boolean Example::save_state ( ObjectState& os, ObjectType )
{
    return (os.pack(A) && os.pack(B) && os.pack(C));
}
```

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 `ObjectState` 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` (shown below) 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 *Arjuna* 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.

```
enum LockResult
{
     GRANTED, REFUSED, RELEASED
};

enum ConflictType
{
     CONFLICT, COMPATIBLE, PRESENT
};

class LockManager : public StateManager
{
public:
     LockResult setlock (Lock *toSet, int, unsigned int);
     . . .

     /* virtual functions inherited from StateManager */

     virtual Boolean restore_state (ObjectState& os, ObjectType ot) = 0;
     virtual Boolean save_state (ObjectState& os, ObjectType ot) = 0;
     virtual const TypeName type () const;

protected:
     /* Constructors and destructor */

     LockManager (ObjectType ot = RECOVERABLE);
     LockManager (const Uid& storeUid);
     ~LockManager ();

private:
     /* non-virtual member functions */

     ConflictType lockConflict (const Lock& otherLock);
     . . .
};
```

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 *Arjuna* are not special system types. Instead they are simply instances of other *Arjuna* 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.

```
extern const LockMode READ;
extern const LockMode WRITE;

enum LockStatus
{
     LOCKFREE, LOCKHELD, LOCKRETAINED
};

class Lock : public StateManager
{
public:
     /* Constructors and destructor */

     Lock (LockMode lm);                    /* Lock constructor */
     virtual ~Lock ();

     /* virtual member functions */

     virtual Boolean conflictsWith  (const Lock& otherLock) const;
     virtual Boolean modifiesObject () const;

     /* inherited functions */

     virtual Boolean restore_state (ObjectState& os, ObjectType ot);
     virtual Boolean save_state (ObjectState& os, ObjectType ot);
     virtual const TypeName type () const;
     . . .
private:
     . . .
};
```

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.

```
class AtomicAction : public StateManager
{
public:
     AtomicAction ();
     virtual ~AtomicAction();

     static AtomicAction *Current ();

     Boolean add (AbstractRecord *);
     ActionStatus status();
     Boolean isAncestor (const Uid&);
     AtomicAction *parent ();

     virtual ActionStatus Abort ();
     virtual ActionStatus Begin ();
     virtual ActionStatus End ();
     . . .
protected:
     Boolean phase2Commit ();
     Boolean phase2Abort ();
     PrepareOutcome prepare ();
     . . .
private:
     RecordList *const pendingList;
     RecordList *const preparedList;
     RecordList *const readonlyList;
     . . .
};
```

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.

```
class AbstractRecord : public StateManager
{
public:
     virtual ~AbstractRecord ();

     . . .              // various management operations

     virtual Boolean nestedAbort () = 0;
     virtual Boolean nestedCommit () = 0;
     virtual PrepareOutcome nestedPrepare () = 0;
     virtual Boolean topLevelAbort () = 0;
     virtual Boolean topLevelCommit () = 0;
     virtual PrepareOutcome topLevelPrepare () = 0;

protected:
     AbstractRecord (const Uid&);
     . . .
private:
     . . .
};
```

Program 10: class AbstractRecord

Thus, when an action is committed by the user (using the `End` 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), distribution (RajdootCallRecord) and object lifetime (ActivationRecord)).

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

```
#include "AppointMent.h"

// The following stub specific commands are actually the default
// @Remote, @NoMarshall
class Diary : public LockManager
{
public:
     Diary(ArjunaName AN);
     ~Diary();

     String WhereIs(time_t now, String user);

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

     virtual Boolean save_state(ObjectState&, ObjectType);
     virtual Boolean restore_state(ObjectState&, ObjectType);
     virtual const TypeName type() const;

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

Program 11: class Diary

The GetNextAppointment operation of this class could be written as shown in Program 12. The *Arjuna* 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 *Arjuna* 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.

```
AnAppointment Appointments::GetNextAppointment( time_t time )
{
     AtomicAction A;
     AnAppointment entry;

     A.Begin();
     if (setlock(new Lock(READ), RETRIES) == 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.End();
     }
     else
          A.Abort();
     return entry;
}
```

Program 12: Example User Code

4.4. Coping With Distribution

In order to cope with distribution, we need to provide the necessary facilities for the distributed execution of atomic actions. This has three aspects:

(i) Provision of the necessary object support infrastructure required for the management of objects by servers.

(ii) Addition of distributed atomic action capability by enhancing the atomic action and abstract record classes.

(iii) Interfacing to the underlying RPC system.

4.4.1. Object Support Infrastructure for Distribution

In the present version of *Arjuna*, the object support infrastructure used is very simple. Basically, it consists of the two processes mentioned earlier, namely a store daemon and a connection manager; in addition, there is a housekeeper process at each node that monitors liveness of client nodes and implements orphan detection and killing. Top-level actions access remote objects through independent servers. On receiving an `initiate` request from a client, the connection manager simply forks a (server) process supplying it the name

of the binary file containing the object specific server code. This server first checks if a server for the client already exists on the node, and if so, it returns the communication identifier of the existing server to the caller and dies; otherwise it becomes a fully fledged server by executing the named binary file. Thus *n* servers for an object can exist at a node if *n* top-level actions are sharing that object. The concurrency controller for the object will ensure proper sharing: for example, only one server will ever be allowed to modify the state of the object. This simple scheme has the important advantage of providing isolation between independent actions, but is inefficient for frequently shared objects. A better approach would be to provide a flexible scheme for managing objects, permitting a server to be shared if required (this will naturally require multi-threading), and even allowing a server to manage instances of several different classes of objects (this requires a dynamic loading facility). Such options were discounted in this version of *Arjuna* because at the time the implementation started (late eighties) there were no satisfactory, portable threads package available for UNIX, nor was there any dynamic loading facility available. Future versions of *Arjuna* will be based on a more flexible object support infrastructure.

4.4.2. Implementing Distributed Atomic Actions

For the atomic action system to function efficiently and correctly in a distributed environment it is important that any atomic actions active in a client are correctly reflected in any remote object server that the client has accessed. This ensures that remote objects are correctly managed should the atomic action in the client commit or abort. These server-side atomic actions, (termed *server actions* and implemented by the class `ServerAtomicAction`) behave solely as representatives of any actions in the client and are created automatically by the system as needed. The class `ServerAtomicAction` is derived from `AtomicAction` and essentially behaves identically to it. The primary difference between the two classes is that `ServerAtomicAction` allows each of the operations that constitute the two-phase protocol to be explicitly called - which `AtomicAction` does not. This is important since these operations will actually be invoked directly by the commit or abort of an action in the client.

Since the action hierarchy present in the client must be reflected in the server it must be propagated somehow. This propagation is handled by classes that interface with the underlying RPC mechanism. All calls from a client to a server automatically have the current action hierarchy (as a list of action UIDs) prepended to them. When the call is received in the server, this list is extracted and compared with the hierarchy that currently exists. If there is a discrepancy, then the hierarchy in the server is *made to conform* to that sent in the call. This may mean creating new `ServerAtomicActions` (if the client hierarchy is deeper than the one in the server), aborting `ServerAtomicActions` (if the hierarchy is shallower) or a combination of the two.

The use of abstract management records provides a simple handle on the implementation of the distributed two-phase commit protocol which can be implemented using the same technique. That is, the classes that interface to the RPC system create and register RPC management records as calls are made to remote objects. Thus when an action commits or aborts in the client and the record list is processed then the

RPC management records transmit the operation to the appropriate server which invokes the appropriate operation on the current `ServerAtomicAction`.

In the same way that the state of a top level atomic action is saved to the object store at the end of a successful prepare phase, then the state of the corresponding server atomic actions is also saved at the same time. The information saved is different (it includes the name of the co-ordinating node, for example), but its presence in the object store indicates the same thing. That is, this action has prepared successfully.

4.4.3. Interfacing To The Underlying RPC

The underlying RPC system is not normally visible to programmers. Instead a Stub Generation system is employed to create the required code to interface to the RPC mechanism. Stub generation in *Arjuna* is different to that typically employed in other systems in that it does not require the use of a separate Interface Description Language (IDL). Instead the *Arjuna* stub generator is based upon the philosophy that the interface to an object has already been specified (in C++) when the object was originally designed with a non-distributed implementation in mind. To this end the *Arjuna* stub generator accepts as input the standard C++ header files that would normally be input to the C++ compiler. The stub generator places as few demands on the underlying RPC system as it can. In particular it requires only the ability to initiate a connection to some remote server, a means of making actual calls, and a method of breaking the connection. This separation of the details of the actual RPC from the interface seen by the generated stub code is important and has many advantages. In particular, stubs can be generated without regard for the actual RPC mechanism used providing that the RPC mechanism complies with the required interface specification.

The stub generated code uses only three classes: `ClientRpcManager`, `ClientRpc`, and `ServerRpc`. As expected `ClientRpc` represents the client side view of the RPC mechanism and provides operations to initiate an RPC connection (`initiate`), perform a remote call (`call`), and break the RPC connection (`terminate`). `initiate` should establish a binding between the client and the server through whatever mechanism the underlying RPC mechanism provides using the information provided by the `ArjunaName` object supplied as a parameter. `terminate` breaks the binding between a client and a server, while `call` performs the actual RPC. The main parameters to `call` are an opcode indicating which operation to invoke in the server and buffers for the call arguments and returned results, together with two status flags.

```
class ClientRpc
{
public:
     //
     // Fundamental generic operations provided by
     // the RPC interface.
     //

     ClientRpc (ArjunaName* ArjNam);
     ClientRpc (const char* serviceName = 0);
     virtual ~ClientRpc();

     RPC_Status initiate();

     RPC_Status call(Int32 opcode, RpcBuffer& callbuff,
                     Int32& errcode, RpcBuffer& result);

     virtual RPC_Status terminate();
};
```

Program 13: class ClientRpc

Similarly, the server side of the connection is handled by `ServerRpc` which provides operations to receive an incoming request (`getWork`) and return some results (`sendResult`).

```
class ServerRpc
{
public:
     ServerRpc ();
     virtual ~ServerRpc ();

     int initialise (int argc, char *argv[]);
     void getWork (Int32& opcode, RpcBuffer& call);
     void sendResult (Int32 errcode, RpcBuffer& result);
};
```

Program 14: class ServerRpc

Normally the stub generated code does not invoke any of the client side operations directly. Instead this is handled by the third class `ClientRpcManager`. The constructor for this class invokes `initiate`, while the destructor invokes `terminate`. Naturally it exports the `call` operation unmodified. This approach ensures that client/server connection and disconnection is handled simply by creating and deleting instances of the control class.

```
class ClientRpcManager
{
public:
     ClientRpcManager (ArjunaName *);
     ClientRpcManager (ClientRpc * = 0, int = 0);
     ClientRpcManager (const char *, int = 0);
     virtual ~ClientRpcManager ();

     RPC_Status call (Int32, RpcBuffer&, Int32&, RpcBuffer&);

     static ClientRpcManager *createInstance (const char *, int = 0);

private:
     int initiated;
     ...
};
```

Program 15: class ClientRpcManager

4.4.4. Parameter Marshalling

Implementing remote procedure calls inevitably requires a mechanism by which arguments and results can be transferred between the client and the server. This typically involves packing the arguments into a buffer used by the underlying RPC transport mechanism for transmission and then unpacking them again at the receiving machine. These operations are frequently referred to as marshalling and unmarshalling.

C++ operator overloading is used to simplify considerably the code required to marshall (encode) and unmarshall (decode) arguments to and from the underlying RPC buffers. In particular, the operators >> and << have been adopted for this purpose (similar to their use in the C++ I/O system). Thus << is used to marshall arguments into the buffers used by the RPC mechanism, and >> to unmarshall arguments from the buffers regardless of the actual type of the argument. The RPC buffer class (`RpcBuffer`) provides a set of operations that permit the marshalling and unmarshalling of all of the basic types of C++ (int, char, etc.). The marshalling of more complex structures is simply achieved by breaking the structure up into its component parts and marshalling each independently. The actual encoding scheme currently used is the same as that used by the persistence mechanisms in *Arjuna* that enable a C++ object to be stored on disk (that is `RpcBuffer` is derived from the class `Buffer` and uses its `pack` and `unpack` operations directly). There is, however, no reason why some other scheme could not also be used.

4.4.5. Client and Server Classes

For each class declaration that it reads from its input file the stub generator will (when appropriate) generate three new class definitions. These class definitions represent:

(i) The replacement class for use by the programmer in the client application.

(ii) The server stub class responsible for decoding an incoming RPC request, unmarshalling any incoming parameters, invoking the required operation, and marshalling and returning any output values prior to returning control to the caller.

(iii) A renamed version of the original input class that is instantiated in the server as required.

For example, the class definition shown in Program 11 would result in the generation of the definitions and supporting code shown in the following sub-sections.

4.4.5.1. Client Interface

Simple renaming tricks played using the standard pre-processor enable this class to be transparently used under its original name in the programmer's application code.

```
class RemoteDiary : public RemoteLockManager
{
public:
     RemoteDiary (ArjunaName , ClientRpcManager *crpc = 0);
     ~RemoteDiary ();

     String WhereIs (time_t , String );
     AnAppointment GetNextAppointment (time_t );
     int AddAppointment (AnAppointment );
     int DelAppointment (time_t );
     virtual Boolean save_state (ObjectState & , ObjectType );
     virtual Boolean restore_state (ObjectState & , ObjectType );
     virtual const TypeName type () const;

protected:
     RemoteDiary(ClientRpcManager *, const RpcBuffer&, char);

private:
     virtual ClientRpcManager *_get_handle () const;

     ClientRpcManager *_client_handle;
     . . .
};
```

Program 16: class RemoteDiary

This generated client stub class has the same set of public operations as the original (although any constructors have had an extra argument added to them, this is effectively invisible and the code written to use instances of the original class will still compile). Public instance variables, however, are deliberately not included in the generated class for reasons that will be explained in a later sub-section. Internally the implementation of the class is totally different. Firstly, only variables pertinent to the establishment and maintenance of the RPC connection are present. Secondly, all of the operations are re-implemented to perform the appropriate parameter (un)marshalling and RPC invocation. Thirdly, some additional operations are introduced including an additional protected constructor which is used to ensure that certain information pertinent to the RPC system is correctly propagated to the stub generated versions of all base classes (if any).

4.4.5.2. Client Side Code

The generated client stub code for each operation follows a standard pattern: marshall arguments, send invocation, await reply, unmarshall results, and return to caller. The client stub code produced exploits the C++ constructor and destructor notions to ensure that the real (user) objects in the server have lifetimes that match the lifetime of the (stub) objects in the client. At the point that the stub object enters scope in the client (and thus the constructor operation of the object is automatically executed) then binding of client to server is accomplished using the supplied `ArjunaName`. Furthermore, the first RPC sent to the newly created server corresponds to the invocation of the constructor for the real object and is passed the arguments presented by the client application. Similarly, when the stub object is destroyed in the client, the generated destructor causes an RPC request to be sent to the server causing the execution of the remote object destructor. The server is destroyed when the `ClientRpcManager` instance that performed the initiate request is destroyed.

```
AnAppointment RemoteDiary::GetNextAppointment (time_t now)
{
     /* call and return buffers */
     RpcBuffer rvBuffer;
     RpcBuffer callBuffer(_myHashVal);
     RpcBuffer replyBuffer;
     RPC_Status rpcStatus = OPER_UNKNOWN;
     Int32 serverStatus = OPER_INVOKED_OK;
     AnAppointment returnedValue;
     /* marshall parameter */
     callBuffer << now;
     /* do call */
     rpcStatus = _clientHandle.call(31096804, callBuffer, serverStatus, replyBuffer);
     if ((rpcStatus == OPER_DONE) && (serverStatus != DISPATCH_ERROR))
     {
          switch (serverStatus)
          {
          case OPER_INVOKED_OK:
               replyBuffer >> rvBuffer;
               rvbuffer >> returnedValue;
               break;
          default:
               _clientHandle.rpcAbort();
          }
     }
     else
          _clientHandle.rpcAbort();
     return (returnedValue);
}
```

Program 17: Stub Generated Client Code for GetNextAppointment

The method for generating the server side interface and code follows a similar pattern and is not discussed here; more complete details can be found in [Parrington95].

4.5. Naming and Binding Service

The Naming and Binding service together supports the location of objects by name and the management of naming contexts. Such services are often designed as a part of a name server which becomes responsible for mapping user supplied names of objects to their locations. Like the RPC and Object storage services, *Arjuna* can exploit any available name server, or a data storage service for storing naming and binding information. The default system itself has been implemented out of persistent *Arjuna* objects, thereby providing atomic access to naming and binding information. The atomic naming and binding system is used for supporting object replication (to be discussed in a subsequent section).

Arjuna provides an abstract naming scheme for *persistent* objects. This naming scheme is based around the use of instances of the class `ArjunaName`. The `ArjunaName` class provides an interface to a (possibly replicated) naming service, allowing the names of persistent objects to be registered with the name service and searched for. Using this scheme, individual persistent objects (whether local or remote) can be named by strings and the naming service can be accessed from anywhere in the distributed environment.

```
class ArjunaName
{
public:
     ArjunaName ();
     ArjunaName (const char* Name);
    ~ArjunaName ();

    Boolean lookUp ();
    Boolean registerName ();
    Boolean registerName (const char*, const StateManager&);

    void setObjectName (const char*);
    void setObjectUid (const Uid&);
    void setServiceName (const char*);
    void setHostName (const char*);

    const char* const getObjectName ()const;
    Uid   getObjectUid ()const;
    const char* const getServiceName () const;
    const char* const getHostName () const;
};
```

Program 18: class ArjunaName

The `ArjunaName` entry held within the name service contains all of the necessary information for locating and using the persistent object, e.g., the hostname of the node where the server is located, its service name, and the `UID` of the persistent object.

The `lookUp` method contacts the name service to attempt to retrieve the stored object information which matches the completed fields within the `ArjunaName` object. The information returned can be obtained using the various `get` methods. The `registerName` methods operate in a similar manner but build the actual entry to be added to the name service.

```
#include <ArjServers/ArjName.h>
. . .
class Example : public LockManager
{
public:
  Example(char* name);
  Example(ArjunaName arjunaName);
  . . .
private:
  . . .
};
```

Program 19: Example ArjunaName Usage

To use an `ArjunaName`, a persistent object's interface is augmented with two additional constructors. The first constructor (with `char*` parameter) is for creating a *new* persistent object, the name of the new persistent object being the value of the parameter (e.g., "`MySpreadSheet`"). The second constructor (with `ArjunaName` parameter) is for accessing existing (already created and registered with the name server) persistent objects.

When the `ArjunaName` instance is created it contacts the name server to obtain the remaining information about the persistent object named in the constructor. If this is a remote object then this information includes the hostname of the remote object as well as the persistent object's `UID`.

4.6. Crash Recovery

The *Arjuna* crash recovery mechanism is built around the fact that when a top level action 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.

Crash recovery is actually driven from the server end rather than the original client. This is because the information saved by server atomic actions includes the name of the co-ordinating client node. Thus servers know their clients, but clients do not necessarily know all of their servers. The crash recovery process therefore consists of scanning the object store for saved server atomic action states (that is server actions that had prepared but not committed or aborted) and for any found querying the co-ordinating node for the final outcome of the action. Once the commit or abort decision has been completed the server atomic action entry is deleted from the object store.

The query process at the co-ordinating node uses the presence (or absence) of a corresponding atomic action entry in the object store to determine whether the client action committed or aborted. The system works in a *presumed abort* mode, that is, if no record exists then the action is aborted. This arises from the fact that the record is written only at the end of a successful prepare phase implying that all servers 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 server actions whose state is unknown.

4.7. Advanced Features

4.7.1. Replication

The availability of *Arjuna* objects can be increased by replicating them on several nodes, for example, by storing their states in more than one object store. Object replicas must be managed through appropriate replica-consistency protocols to ensure that object copies remain mutually consistent.

A replication technique could either be *active* or *passive*. In active replication, more than one copy of an object is activated on distinct nodes and all activated copies perform processing. Passive replication in its basic form requires maintaining multiple copies of persistent states but only a single copy (primary) is activated; at commit time, the activated copy checkpoints its state to all the object stores where the states reside. One of the advantages of this form of passive replication is that it can be implemented without recourse to reliable group communication that is required for active replication (as only one replica is activated at any time). So passive replication can be supported on top of any conventional RPC system. Since *Arjuna* is intended to be capable of running on top of commonly available distributed computing platforms, the default replication scheme employed for *Arjuna* objects is passive. However, active replication of *Arjuna* objects is also

possible [Little90]. Indeed, we have implemented the necessary infrastructure within *Arjuna* to enable it to support both active and passive replication of objects.

This infrastructure consists of a naming and binding service for persistent replicated objects that ensures that applications only ever get to use mutually consistent copies of replicas. The naming and binding service itself has been implemented out of persistent objects whose operations may be invoked under the control of atomic actions; that is how the service is capable of providing the necessary consistency. Further details are available in [Little93, Little94].

4.7.2. Object Clustering

Arjuna also provides a dynamic performance improvement scheme that is based on controlling clustering of persistent objects [Wheater94]. There are two mutually conflicting factors that influence the execution time of atomic actions:

(i) The disc I/O time taken to activate objects (load their states from object stores) and then deactivate them (copy their modified states back to object stores) at commit time.

(ii) Time taken up because of access conflicts: an action operating on an object could block the progress of other actions requesting access to that object (e.g., due to a read-write conflict).

The disc I/O time can be reduced by decreasing the potential number of object activations/deactivations: it is faster to activate (deactivate) a single object of size N disc blocks than to activate (deactivate) n, $n>1$, different objects of total size N (this is because like most object stores, the *Arjuna* object store stores the state of an object in contiguous disc blocks). On the other hand, access conflicts can be reduced by decomposing an application's state into large number of persistent objects each independently concurrency controlled.

The access pattern and the degree of sharing can vary during the lifetime of applications. There could be a phase when applications require access to a large number of related objects, but object sharing is infrequent; during this phase, clustering related objects into a few large objects for the purposes of activation/deactivation would improve performance. Similarly, there could be a phase when applications need to share objects frequently, then declustering objects into a number of independently concurrency controlled units would be better. Our scheme provides a means of dynamically reconfiguring storage structures of application objects, from no clustering (all objects treated as independent), to maximum clustering (all objects grouped together), including any intermediate level of clustering to suit a given pattern of usage. Any such configuration is carried out using atomic actions, so the configuration changes take place atomically, even if failures occur.

5. Performance

To give an indication of the performance of the prototype system a series of performance tests have been carried out on a Sun SPARCstation 10/30 running Solaris 2.3. The machine was normally loaded during the performance evaluations and was running all of the standard daemons in order to simulate the typical environment in which *Arjuna* operates. All of the performance evaluation tests involved operations upon a number of objects each of which was 1024 bytes in size. The times were measured using standard system calls and represent elapsed time.

Analysing the performance of *Arjuna* applications is complex since it depends on many external and internal factors. For example, one key external factor affecting performance is the effect of the kernel file buffer cache. If an object has been accessed recently such that its state is still in the buffer cache then the activation time for the object is significantly reduced since physical disk I/O is unnecessary. Similarly, the object store file descriptor cache which attempts to reduce the number of `open` system calls can significantly affect performance depending upon the number of different objects accessed.

5.1. Local Transactional Overhead

Any transactional system must add some measure of overhead to an application. In *Arjuna* most of this overhead occurs due to the additional system calls the system makes on behalf of the application (time spend executing other *Arjuna* code in user space is tiny in comparison). Additionally read-only actions have different performance characteristics to those that also modify objects. Experience has shown that the primary factors affecting performance are, not suprisingly, those that involve disk activity, followed by those operations performed by the concurrency controller.

5.1.1. Object Store Overhead

Arjuna applications access the object store for three reasons: reading/writing the state of an object, and action intention list processing. This accounts for the main difference in the characteristics of read-only and write actions since the latter type of action makes many more file system accesses. Note that *Arjuna* only writes to the file system on top-level commit so these differences are only apparent then.

Object Activation. This is a complicated operation requiring location of the object in the object store (via `read_committed`) and its unpacking into a usable C++ object (via `restore_state`). Of these two it is the former that dominates. The operation `read_committed` can map into the following UNIX system calls: `open`, `fcntl`, `fstat`, `lseek`, `readv fcntl`, `close`. The `open` and `close` calls may not actually be made depending upon the effect of the open file cache mentioned earlier.

Object Deactivation. The inverse of the above, requiring a `save_state`, followed by `deactivate`. Here `deactivate` maps into: `open`, `fcntl`, `writev`, `fcntl`, `close`. Typically the `open` call is not

actually made due to the use of the open file cache and the majority of time is spent waiting in the kernel on the `writev` call (which is forced to be synchronous since the store is required to be stable).

Intention List Processing. When a top-level action commits a list containing information about all of the objects modified by the action must be saved. In the current implementation this is stored in a set of memory mapped files which must be flushed to disk using `memcntl`.

5.1.2. Concurrency Control Overhead

Arjuna does not require concurrency control information to be stable, so most of the concurrency control overhead comes from those system calls that enforce mutual exclusion on the shared memory region in which the locks are stored - notably `sigprocmask` and `semop`.

5.1.3. Performance Evaluation

In order to find the dominant system calls, both top-level read-only and top-level write tests were monitored with *Quantify* to determine the most significant (from the point of view of elapsed time) system calls executed by *Arjuna*. The results are presented below.

Top-Level System Calls - Write Action

For any top-level action that modifies objects the system calls that dominate the elapsed time are (numbers in brackets indicate the number of calls made):

Per Action: `memcntl`(2), `fstat`(13), `getpid`(2), `sigprocmask`(7), and `semop`(4). The first three calls are used in intention list processing, the latter two are used to enforce mutual exclusion and prevent signal reception during critical sections of code.

Per Object: `readv`(3), `lseek`(5), `fcntl`(6), `writev`(2), `fstat`(3), `getpid`(4), `sigprocmask`(6), and `semop`(4). As above the latter two calls are from enforcing mutual exclusion and preventing signals. The other calls are all incurred in reading and writing the state of an object form and to the object store.

In this scenario, the elapsed time is actually dominated by only two calls: the `writev` (to write an object state) and `memcntl` (intention list handling) which together represent ~70% of the elapsed execution time of the test.

Top-Level System Calls - Read Only Action

For any top-level action that only reads objects the following system calls are dominant:

Per Action: `sigprocmask`(7), `getpid`(2), `sigaction`(3), and `time`(1). These are concerned solely with mutual exclusion and signal prevention. Note that since the action is read only no intention list handling calls are needed. The call of the `time` system call is from Uid generation.

Per Object: `readv`(1), `lseek`(1), `fcntl`(2), `fstat`(1), `memcpy`(64), `getpid`(3), `time`(3), `sigprocmask`(4) and `semop`(4). The first five are from object store access; the next two are Uid generation, and the last two are the familiar mutual exclusion/signal handling calls.

Here, since the disk accesses are far fewer, more system calls show up and their distribution is more even in that even the most heavily used (`sigprocmask`) only represents 15% of the total elapsed time.

Analysis

One interesting characteristic of the above traces is that despite the fact that many objects were accessed in the tests the `open` and `close` system calls do not figure in the top time consuming calls. This shows that the store's open file cache is operating effectively. When the cache is disabled `open` becomes far more significant and rivals `writev` for time consumed by top-level write actions. The table below shows the average elapsed time taken by the system calls mentioned above in the test environment:

System Call	Average Elapsed Time (ms)
`writev`	14.0
`memcntl`	21.5
`fcntl`	0.17
`fstat`	0.06
`readv`	0.21 (cached in kernel) 3.4 (uncached)
`semop`	0.08
`sigprocmask`	0.01
`lseek`	0.02
`getpid`	0.02
`time`	0.01
`sigaction`	0.04

Table 1: System Call Performance

Using this information enables the approximate performance of an *Arjuna* application to be calculated. Since nested actions do not perform any intention list processing they effectively have performance characteristics similar to a read-only top-level action.

Given these timings and system call patterns it can be seen that the performance of any *Arjuna* application will be effectively proportional to the number of objects modified by top-level actions which commit since it is only in this case that the very expensive system calls `writev` and `memcntl` are executed.

The next set of performance figures effectively confirm this observation. They represent the performance of the action system, for both nested and top-level actions, controlling operations which either

examine or modify an object. In the nested case only the nested action manipulates the object - the top-level action is simply a wrapper.

	Read Only	Write
Top-level atomic action which commits	9.5 ms	101.0 ms
Top-level atomic action which aborts	9.5 ms	10.5 ms
Nested atomic action which commits (exploiting caching)	5.0 ms	5.5 ms
Nested atomic action which aborts	8.6 ms	9.5 ms

Table 2: Local Atomic Action Performance

As can be seen from the figures above a top-level atomic action that modifies an object, and then commits those modification is a long operation. This is due to need to create, write and then remove an intentions list for each top-level atomic action in addition to the object storage overheads. As outlined earlier these costs are dominated by the execution of the 2 x writev (~28ms), and 2 x memcntl (~43ms) calls which togethor contributed 70% of the elapsed time confirming earlier comments.

The figures obtained for nested actions illustrate one of the effects of caching. Once the first nested action commits, information (in particular the activation state of the object) is propagated to the parent top-level action where it is retained. Thus future nested actions proceed without further reference to the object store. In the abort case all information is discarded requiring that each nested action reactivate (i.e. reload from the store) the object thus the performance is similar to that of a read-only action.

5.2. Distribution Overhead

The following set of figures are an evaluation of the performance of the basic distribution infrastructure:

Server initiation	233.0 ms
Server termination	4.0 ms
Null RPC round trip	3.0 ms

Table 3: RPC Performance

Server initiation requires an RPC to the manager process on the remote node, which in turn will execute two `fork` and two `exec` system calls prior to returning. The rational behind the double `fork` is to ensure that the child process is completely divorced from the manager and the double `exec` occurs because a duplicate server checking process is exec'd first before the required server. The time taken for server initiation is greater than the time to perform a simple double `fork` and `exec`, which required 96.0 ms, due to server

registration. However, server initiation and termination are expected to be fairly infrequently executed operations

The last set of performance figures where obtained to evaluate the performance of distributed atomic action processing, including distributed two-phase commit assuming that a server has already been created to avoid server startup/shutdown costs.

	No lock	**Read lock**	**Write lock**
Distributed top-level atomic action which commits	5.4 ms	19.0 ms	130.0 ms
Distributed nested atomic action which commits	5.3 ms	9.0 ms	10.0 ms

Table 4: Distributed Atomic Action Performance

The results show several things. Firstly, even a null action (i.e., one that does not touch any object) imposes overheads over the Null RPC round trip time. This is due to the overhead involved in transmitting and then building the action context in the server. Secondly, the performance of the distributed system is approximately equivalent to the non-distributed case with the addition of the extra RPCs for the call and those send during execution of the two-phase commit protocol. For example, in the read-only case two RPCS are sent (~10.8 ms) which in combination with a read-only action time (~9.5 ms) combine to yield the final result (read-only operation do not need to participate in phase-2, hence only two RPCs are sent).

5.3. Final Comments

One immediately obvious problem shown by the above results is that the cost of accessing the object store is quite high. This was only to be expected since the object store implementation sits directly on top of the UNIX file system. Furthermore, *Arjuna* uses the object store not only to store user objects but also for atomic action commit information to ensure the permanence of effect properties. Thus it treats the store as stable and in order to ensure this performs all of the I/O operations in synchronous mode to ensure that the UNIX kernel does not buffer any critical data internally. This has a particularly bad effect on top-level actions which commit where not only does the user's object have to be written but also the atomic action intentions list as well. We continue to refine the object store to reduce this bottleneck. It is worth noting that commit times for write actions can be reduced significantly if stable main memory is made available; expected times then would resemble read commit time (of the order of 10 msecs).

6. Retrospective Examination

Arjuna has proved to be a useful research tool. It has shown that it is possible to build a viable reliable distributed applications toolkit without modifying language or underlying system. A major benefit of this

approach is that application builders can use *Arjuna* in conjunction with other existing tools. For example, to construct user interfaces to *Arjuna* applications, we often make use of the publicly available Interviews Toolkit.

Since its release in 1992, we have strived hard to maintain and improve the system (thanks to comments and bug reports from numerous users), keeping track of changes to C++ and its compilers and UNIX systems from different sources. *Arjuna* has been used regularly by us for teaching distributed computing to undergraduate and graduate students. In addition, it has been used for a variety of other purposes. This has given us useful insights into the strengths and weaknesses of *Arjuna* as well as C++. In light of this, we first examine how effectively we have met the stated design goals of *Arjuna* (modularity, integration of mechanisms and flexibility) and then discuss the suitability of C++ for programming distributed applications.

6.1. A Critical Look at Arjuna

We begin by briefly describing some of the areas in which *Arjuna* has been used.

(i) *Object oriented distributed database system*. *Arjuna* has been used for building an experimental distributed database system called *Stabilis* [Buzato92]. *Stabilis* is currently being used in one experiment as an information base for a run-time management system for distributed applications and in an another experiment as a repository for an Internet information discovery tool.

(ii) *A Student Registration system*. The management information services unit of our University, responsible for computerisation of University's administration, has collaborated with us for building a student registration system. In the past, all students went to a central registration office for registering themselves with the University. In the new system, admission and registration is carried out at the departments of the students. The system has a very high availability requirement: admissions tutors and secretaries *must* be able to access and create student records (particularly at the start of a new academic year when new students arrive). *Arjuna* offers the right set of facilities for this application, enabling the University to exploit the existing campus network and workstation clusters to provide transactional access to highly available data. The current configuration uses eight high performance workstations that together serve about eighty user terminals scattered over the campus; three of the workstations are used exclusively for maintaining a triplicated objectstore.

(iii) *Provision of atomic transactions to an existing programming system*. A software company that specialises in providing data management software to companies in oil and financial sectors is integrating the *Arjuna* class libraries into its own software product for obtaining atomic action functionality. This exercise involves replacing the default RPC mechanism of *Arjuna* by that of DCE, and replacing the existing object store by a commercial object store that is used within the company's product.

(iv) *Fault-tolerant Parallel programming over a network of workstations*. Use of networks of workstations for obtaining speedups by running parallel versions of programs (e.g., floating point matrix

calculations, computational fluid dynamics calculations) is gaining widespread popularity amongst scientists. Many such scientific parallel programming applications need to manage large quantities of data (requiring several megabytes of storage) and parallel calculations may last a long time (from hours to days). Such applications could benefit from adequate support for data persistence and fault-tolerance. We have explored the use of *Arjuna* for this type of applications, and have been encouraged from our initial results [Smith95].

We now take a critical look at *Arjuna*, examining in turn its modularity, integration of mechanism and flexibility features.

Modularity. By encapsulating the properties of persistence, recoverability, shareability, serialisability and failure atomicity in an Atomic Action module and defining narrow, well-defined interfaces to the supporting environment, *Arjuna* does achieve a significant degree of modularity. The structure of *Arjuna* makes it is possible to replace default RPC, Object Store, Naming and Binding modules by the modules of the host system. This makes the system quite portable [Shrivastava94a]. One of our ports has been on to the ANSAware distributed computing platform [ANSA91]. This platform provides RPC, object servers (known as *capsules*) and naming and binding service via a subsystem known as the *Trader*. The port was performed by mapping the RPC operations (`initiate`, `terminate` and `call`) onto those provided by ANSAware. Application (iii) mentioned above actually requires simultaneously discarding all of the three default modules of *Arjuna* in favour of those of the host environment. We have performed a feasibility study to ascertain that this is indeed possible.

The current design for *Arjuna*, while elegantly sorting out the functions of the Atomic Action module into classes, nevertheless fails to cleanly separate the interfaces to the supporting environment. Some components are responsible for too many things. For example, the `StateManager` class maintains not only state information but also the unique identifier of the object. The present RPC facility, while supporting the interface discussed, is also responsible for the creation of object servers, and failure detection and cleanup. These three facilities should have been separated. Notwithstanding these observations, we believe that on the whole, *Arjuna* has met the goal of modularity.

Integration of Mechanisms. Although *Arjuna* does make it relatively easy for programmers to build distributed applications using objects and actions, in many respects the programmer still has to be aware of several additional complexities, for example, writing `save_state` and `restore_state` routines and setting the appropriate locks in operations. Experience has shown that users have not found these complexities as insurmountable problems. These latter problems can be actually alleviated by enhancement of the stub generation system. For example, `save_state` and `marshall` are essentially the same operations, and the latter can already be generated. Furthermore, sufficient syntactic information already exists in the declaration of a member function (so called *const member functions*) to determine whether a read or write lock is needed. Thus by relatively minor enhancements the programmer's burden can be reduced further, provided that the

default behaviour provided by the system is acceptable. If such a scheme is implemented, mechanisms will also be needed for those cases where the programmer wishes to override the default actions.

Flexibility. We have learned our biggest lessons in the area of flexible system structuring. When we began the design of *Arjuna*, we took flexibility to mean that application builders should be able to perform application specific enhancements to the *class hierarchy*, permitting for example, type-specific concurrency and recovery control to be easily produced from the existing default ones. Although the single inheritance hierarchy is somewhat constraining in what is achievable, *Arjuna* does permit such enhancements to be performed. Experience from helping users building *Arjuna* applications has taught us that this is however not sufficient; serious application builders need *controlled access* to the deepest internals of the system! The reason in most cases is for tuning the system behaviour for obtaining higher performance; sometimes this is also because users have some system structuring constraints imposed by existing applications (e.g., where to place object servers and a limit on their numbers), that can only be satisfied by modifications to the internals of *Arjuna*. We give a few representative examples which we hope will be of interest to the readers of this paper.

(i) *Transport mechanism for RPC*. For efficiency reasons, the default RPC mechanism of *Arjuna*, like several other RPC mechanisms, uses connectionless datagrams (UDP) for transporting RPC messages, performing its own fragmentation and reassembly of packets. This has proved adequate in all the applications we have encountered, except application (iv). This application frequently requires RPC messages to carry several megabytes of data, for which the existing protocol proved inadequate (as it was not designed for bulk data transfer). We changed *Arjuna* RPC to enable it to use a connection based transport layer (TCP) where needed. What is required is for RPC to be able to choose the most appropriate transport mechanism at run time.

(ii) *Server management*. As stated before, *Arjuna* employs a simple scheme for accessing remote objects: Top-level actions access remote objects through independent servers. Although we knew that this will not be an adequate policy under all circumstances, our experience has indicated that no single policy is adequate. Most applications require some control over object to server binding, including enabling and disabling of caching of objects at clients. For example, application (iii) requires caching of *read-only* objects; we have met this by making specific changes to *Arjuna*. The database application, Stabilis, routinely handles large numbers of objects, and therefore for performance reasons needs to minimise the number of servers. The designers of Stabilis have done this by implementing their own server multiplexing scheme, a complication they would have preferred the system to handle.

(iii) *Concurrency control*. The current design of *Arjuna* constraints the concurrency controller of an object to be co-located with the servers. A more flexible scheme would permit the concurrency controller of an object to be remote from the activated copies of the object. Such a scheme can then be used by clients of application (iii) for more flexible form of object caching, permitting multiple copies of an object for read access and a single copy for write access that is allowed to invalidate the remaining

copies. In the absence of such a facility, the builders of Stabilis database system implemented there own object caching scheme on top of *Arjuna*.

In summary, users have managed to build non-trivial applications using *Arjuna*. Occasionally, meeting their requirements have required making changes to *Arjuna*. It is an indication of the sound design of the system that it permits such changes to be made. Nevertheless, such changes have required intervention of the designers of the system. We are currently exploring new system structuring techniques that will permit (expert) system builders to perform such changes themselves.

6.2. An Examination of C++

In retrospect our choice of C++ has been both beneficial and responsible for some of our problems. On the positive side it has enabled us to make *Arjuna* available to a wide audience both academic and commercial. On the negative side the primary stumbling block has been the implicit assumption in the language of the existence of a shared address space which causes some features of the language to be impossible to extend to the distributed environment. Examples of such constructs include:

(i) *Variable length argument lists*. These cannot be marshalled automatically since the stub generator cannot determine at the time it processes the header file how many arguments will need to be marshalled on any given call.

(ii) *Public variables and friends*. These break the assumed encapsulation model and allow potentially unconstrained access to the internal state of an object. Since that object may now be remote from the client application such variables will typically not exist or at least not be accessible in the same address space.

(iii) *Static class members*. C++ semantics state that only a single copy of a static class variable exists regardless of the number of instances of the class in existence. These semantics cannot be enforced in a distributed environment since there is no obvious location to site the single instance, nor any way to provide access to it.

Additional problems arise from the fact that a C++ class declaration describes not only the interface to a class but also much of its implementation. Furthermore the language continues to evolve requiring the stub generator to evolve with it as new language features are added. These points have compromised the stub generation process *Arjuna* uses to achieve distribution and have the effect of lowering the overall transparency to the programmer. However, with care, applications can be written that are location and access transparent, requiring only minimal additional programmer assistance for acquiring fault-tolerance and persistence capabilities. However, the point remains that stub generation relieves the programmer of a significant proportion of the burden involved in the distribution of applications (see [Parrington95] for more details).

Finally, since the original design of *Arjuna* was conceived under an early version of C++ the design was based upon the use of single inheritance only. Whether the use of multiple inheritance would be an advantage or not remains open at this time. Other features such as exception handling and templates would certainly have changed the way parts of the system are implemented but until compilers exist that implement these features reliably we are reluctant to make use of them.

7. Future Directions

Work on *Arjuna* is proceeding in two directions, one with a relatively short term goal and the other potentially long term. On a short term basis, we are examining how the current version can be made to comply with the Object Transaction Service (OTS) standard currently being defined by the OMG[OMG94b]. This extension, in conjunction with several other extensions such as the concurrency control service[OMG94a], are likely to become the standard for transaction processing systems of the future. A simple comparison reveals that *Arjuna* already effectively implements all of OTS and its related services but with different interfaces. However, the mapping between the two is straightforword. With minor modifications the `LockManager` and `Lock` classes can be used to implement the concurrency control service; `AtomicAction` effectively implements the 'Current' interface from the OTS, and `Buffer` can similarly be used to provide the externalisation service[OMG94d].

The main aspect of our long term goal concerning *Arjuna* is to restructure it in a way that will provide controlled access to its internals in a manner that will permit users to customise the implementation to suit the specific needs of their applications [Shrivastava94b]. The basic approach to structuring that we are adopting has some resemblance to the *open implementation* approach suggested in [Kiczales92]. Kiczales proposes a way of system structuring in which abstractions do expose their implementations. Such open implementations have two interfaces per implementation: the *traditional* interface and an *adjustment* interface (the meta-level interface) that provides a way of tuning or customising the implementation (see also [Stroud93] for related ideas). We intend to provide a clean separation between the interface of a class and its implementation, permitting multiple implementations for a given interface, with the selection of the most suitable implementation being left as a run-time decision. Each implementation will also provide a control interface (the adjustment interface) for customisation [Wheater95]. An immediately useful application of this will be to support more than one implementation for RPCs, a need that was identified earlier.

In the present version of *Arjuna*, the object support infrastructure employed is very simple, enabling clients to access remote objects through independent servers. This simple scheme has the important advantage of providing isolation between independent actions, but is inefficient for frequently shared objects. As observed earlier, applications often require some control over object to server binding, including enabling and disabling of caching of objects at clients. A better approach therefore would be to provide a more comprehensive object support infrastructure for managing objects, with support for caching and migration, and permitting a server to be shared, and capable of managing more than one object. With this view in mind, we have designed an object

support system that permits control over object servers and also permits clients to create, copy and exchange references to objects. In this system, a client holding a reference is able to invoke operations on that object even if the object has moved to some other location [Caughey93, Caughey95]. Our design scales to systems of arbitrary size and is portable since it only requires a few standard capabilities from the underlying operating system.

Acknowledgements

The current *Arjuna* system is the product of a diverse and varying team over many years. Apart from the authors (some of whom have been with the project since its conception) the following have all made contributions: L. Buzato, A. Calsavara, S. Caughey, G. Dixon, J. Geada, D. Ingham, D. McCue and J. Smith. F. Panzieri, and F. Hedayati were responsible for the implementing the default RPC system of *Arjuna*. The work reported here has been supported in part by grants from the UK Ministry of Defence, Engineering and Physical Sciences Research Council (Grant Number GR/H81078) and ESPRIT project BROADCAST (Basic Research Project Number 6360).

References

ANSA91 ANSA Reference Manual, Cambridge, UK, 1991. (Available from APM Ltd., Poseiden House, Cambridge, UK.)

Balter91 Balter, R. et al, “Architecture and Implementation of Guide, an Object-Oriented Distributed System,” Computing Systems, 4 (1), pp. 31-67, April 1991.

Black87 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.

Birman87 Birman, K. and T. Joseph, “Exploiting virtual synchrony in distributed systems,” in 11th Symposium on Operating System Principles, ACM SIGOPS, November 1987.

Birman93 Birman, K. , "The process group approach to reliable computing", CACM , 36, 12, pp. 37-53, December 1993.

Buzato92 Buzato, L.E., and A. Calsavara, “Stabilis: A Case Study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects,” Proceedings of the Fifth Int. Workshop on Persistent Objects, San Miniato, Italy, September 1-4, 1992.

Campbell93 Campbell, R.H., N. Islam, D. Raila and P. Madany, “Designing and Implementing Choices : An Object-Oriented System in C++,” Communications of the ACM, Vol. 36, No. 9, September 1993.

Caughey93 Caughey, S. J., G. D. Parrington and S. K. Shrivastava, “SHADOWS - A Flexible Support System for Objects in Distributed Systems,” Proc. of 3rd IEEE Intl. Workshop on Object Orientation in Operating Systems, IWOOOS93, pp. 73-82, Asheville, North Carolina, December 1993.

Caughey95 Caughey, S. J. and S. K. Shrivastava, “Architectural support for mobile objects in large scale distributed systems,” 5th IEEE Intl. Workshop on Object Orientation in Operating Systems, IWOOOS95, Lund, August 1995.

Dasgupta91 Dasgupta, P. et al., “Distributed Programming with Objects and Threads in the Clouds Systems,” Computing Systems, Vol. 4, No. 3, pp. 243-275, Summer 1991.

Detlefs88 Detlefs, D., M.P. Herlihy, and J.M. Wing, “Inheritance of Synchronization and Recovery Properties in Avalon/C++,” IEEE Computer, vol. 21, no. 12, pp. 57-69, December 1988.

Dixon87 Dixon, G. N. and S. K. Shrivastava, “Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems,” Proceedings of Sixth Symposium on Reliability in Distributed Software and Database Systems, pp. 107-114, Williamsburg, March 1987.

Dixon89 Dixon, G. N., G. D. Parrington, S. K. Shrivastava, and S. M. Wheater, “The Treatment of Persistent Objects in Arjuna,” Proceedings of the Third European Conference on Object- Oriented Programming, ECOOP89, pp. 169-189, University of Nottingham, July 1989.

Kiczales92 Kiczales, G., “Towards a New Model of Abstraction in Software Engineering,”, Proc. of Intl. Workshop on Reflection and Meta-Level Architecture, Tama-City, Tokyo, November 1992.

Liskov88 Liskov, B., “Distributed Programming in Argus,” Communications of the ACM, vol. 31, No. 3, pp. 300-312, March 1988.

Little90 Little, M. C. and S. K. Shrivastava, “Replicated K-Resilient Objects in Arjuna,” Proceedings of IEEE Workshop on the Management of Replicated Data, pp. 53-58, Houston, Texas, November 1990.

Little93 Little, M. C., D. L. McCue, and S. K. Shrivastava, “Maintaining Information about Persistent Replicated Objects in a Distributed System,” Proceedings of Thirteenth International Conference on Distributed Computing Systems, Pittsburgh, May 1993.

Little94 Little, M. C. and S. K. Shrivastava, “Object Replication in Arjuna,” BROADCAST Project deliverable report, November 1994; available from Dept. of Computing Science, University of Newcastle upon Tyne, UK.

OMG94a Object Management Group, “Concurrency Control Service,” OMG Document 94.5.8, 1994

OMG94b Object Management Group, “Object Transaction Service,” OMG Document 94.8.4, 1994

OMG94c Object Management Group, “Externalisation Service,” OMG Document 94.9.15, 1994

OMG94d Object Management Group, “Persistence Service,” OMG Document 94.10.17, 1994

Panzieri88 Panzieri, F. and S.K. Shrivastava, “Rajdoot: a remote procedure call mechanism supporting orphan detection and killing”’ IEEE Trans. on Software Eng. 14, 1, pp. 30-37, January 1988.

Parrington88 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.

Parrington90 Parrington, G. D., “Reliable Distributed Programming in C++: The Arjuna Approach,” Second Usenix C++ Conference, pp. 37-50, San Fransisco, April 1990.

Parrington92 Parrington, G. D., “Programming Distributed Applications Transparently in C++: Myth or Reality?,” Proceedings of the OpenForum 92 Technical Conference, pp. 205-218, Utrecht, November 1992.

Parrington95
Parrington, G. D., “A Stub Generation System for C++” Computing Systems, Vol. 8, No. 2, to be published, 1995. (Earlier version available as BROADCAST Project deliverable report, November 1994; http://www.newcastle.research.ec.org/broadcast/).

Shapiro89 Shapiro M., et al, “SOS: an Object-oriented Operating System - Assessment and Perspectives,” Computing Systems, Vol. 2, No. 4, pp. 287-338, December 1989.

Shrivastava91
Shrivastava, S. K., G. N. Dixon, and G. D. Parrington, “An Overview of Arjuna: A Programming System for Reliable Distributed Computing,” IEEE Software, Vol. 8, No. 1, pp. 63-73, January 1991.

Shrivastava94a
Shrivastava, S. K., and D. McCue, “Structuring Fault-tolerant Object Systems for Modularity in a Distributed Environment,” IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 4, pp. 421-432, April 1994.

Shrivastava94b
Shrivastava, S. K., “Lessons learned from building and using the Arjuna distributed programming system,” Int. Workshop on Distributed Computing Systems: Theory meets Practice, Dagsthul, September 1994, LNCS 938, Springer-Verlag, July 1995 (also: Broadcast Project Deliverable Report, July 1995, http://www.newcastle.research.ec.org/broadcast/).

Smith95 Smith, J. A. and S. K. Shrivastava, “Fault tolerant execution of computationally and storage intensive parallel programs on a network of workstations: a case study,” Broadcast Project Deliverable Report, July 1995, (http://www.newcastle.research.ec.org/broadcast/).

Stroud93 Stroud, R. “Transparency and reflection in distributed systems,” Operating Systems Review, Vol. 27, pp. 99-103, April 1993.

Wheater94 Wheater, S. M. and S. K. Shrivastava, “Exercising Application Specific Run-time Control over Clustering of Objects,” Proc. of IEEE Second Intl. Workshop on Configurable Distributed Systems, March, 1994, Pittsburgh, pp. 72-81.

Wheater95 Wheater, S. M. and M. C. Little, “The Design and Implementation of a Framework for Extensible Software,” Broadcast Project Deliverable Report, July 95 (http://www.newcastle.research.ec.org/broadcast/).