Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment

Santosh K. Shrivastava
Department of Computing Science,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK.

Daniel L. McCue
Xerox Corporation
Webster, New York 14850, USA.

Abstract

The object-oriented approach to system structuring has found widespread acceptance among designers and developers of robust computing systems. In this paper we propose a system structure for distributed programming systems that support persistent objects and describe how such properties as persistence, recoverability etc. can be implemented. The proposed structure is modular, permitting easy exploitation of any distributed computing facilities provided by the underlying system. An existing system constructed according to the principles espoused here is examined to illustrate the practical utility of the proposed approach to system structuring.

Index Terms

Object-Oriented Systems, Persistent Objects, Migration, Replication, Fault Tolerance, Distributed Systems, Atomic Actions, Atomic Transactions.

The work reported here has been supported in part by grants from the UK Science and Engineering Research Council (Grant Numbers GR/F38402, GR/F06494 and GR/H81078) and ESPRIT projects ISA (Project Number 2267) and BROADCAST (Basic Research Project Number 6360)

1. Introduction

One computational model that has been advocated for constructing robust distributed applications is based upon the concept of using nested atomic actions (nested atomic transactions) controlling operations on persistent (long-lived) objects. In this model, 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 externally visible 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. 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 can then be used to ensure that consistency is preserved even in the presence of concurrent invocations and failures. Designing and implementing a programming system capable of supporting such 'objects and actions' based applications by utilising existing distributed system services is a challenging task. 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. In this paper we present an architecture which we claim to be *modular* in nature: the overall system functionality is divided into a number of modules which interact with each other through well defined *narrow* interfaces. We then describe how this facilitates the task of implementing the architecture on a variety of systems with differing support for distributed computing.

In the next section, we present an 'object and action' model of computation, indicating how a number of distribution transparency mechanisms can be integrated within that model. Section 3 then identifies the major system components and their interfaces and the interactions of those components. The proposed system structure is based upon a retrospective examination of a distributed system - Arjuna [11, 23, 29] - built at Newcastle. The main aspects of this system are presented in section 4 and are examined in light of the discussion in the preceding section. In this section we also describe how the modular structure of the system has enabled us to port it on to a number of distributed computing platforms. The Arjuna system thus demonstrates the practicality of the proposed approach to system structuring discussed in section 3.

2. Basic Concepts and 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 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. We assume that processes on functioning nodes are capable of communicating with each other.

To develop our ideas, we will first describe some desirable *transparency* properties a distributed system should support. It is common to say that a distributed system should be 'transparent' which means that it can be made to behave, where necessary, like its non-distributed counterpart. There are several complementary aspects to transparency [1]:

- **Access transparency** mechanisms provide a uniform means of invoking operations of both local and remote objects, concealing any ensuing network-related communications;
- **Location transparency** mechanisms conceal the need to know the whereabouts of an object; knowing the name of an object is sufficient to be able to access it;
- **Migration transparency** mechanisms build upon the previous two mechanisms to support movement of objects from node to node to improve performance or fault-tolerance;
- **Concurrency transparency** mechanisms ensure interference-free access to shared objects in the presence of concurrent invocations;
- **Replication transparency** mechanisms increase the availability of objects by replicating them, concealing the intricacies of replica consistency maintenance;
- **Failure transparency** mechanisms help exploit the redundancy in the system to mask failures where possible and to effect recovery measures.

As stated earlier, we are considering a programming system in which application programs are composed out of atomic actions (atomic transactions) manipulating persistent (long-lived) objects. Atomic actions can be nested. We will be concerned mainly with tolerating 'lower-level' hardware related failures such as node crashes. So, it will be assumed that, in the absence of failures, 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 failures. 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). Since many object-oriented languages define operation invocation to be synchronous [30], RPC is a natural communications paradigm to adopt for the support of access transparency in object-oriented languages. Furthermore, all operation invocations may be controlled by the use of atomic actions which have the properties of (i) serialisability, (ii) failure atomicity, and (iii) permanence of effect. *Serialisability* ensures that concurrent invocations on shared objects are free from interference (i.e., any concurrent execution can be shown to be equivalent to some serial order of execution). Some form of concurrency control policy, such as that enforced by two-phase locking, is required to ensure the serialisability property of actions. *Failure atomicity* ensures that a computation will either be terminated normally (*committed*), producing the intended results (and intended state changes to the objects involved) or *aborted* producing no results and no state changes to the objects. This atomicity property may be obtained by the appropriate use of backward error recovery, which can be invoked whenever a failure occurs that cannot be masked. Typical failures causing a computation to be aborted include node crashes and communication failures such as the continued loss of messages. It is reasonable to assume that once a top-level atomic action terminates normally, the results produced are not destroyed by subsequent node crashes. This is ensured by the third property, *permanence of effect*, which requires that any *committed* state changes (i.e., new states of objects modified in the atomic action) are recorded on stable storage. A commit protocol is required during the termination of an atomic 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 [5, 13].

The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent objects. In this model, a persistent object not in use is normally held in a *passive* state with its state residing in an object store or object database and *activated* on demand (i.e., when an invocation is made) by loading its state and methods from the object store to the volatile store, and associating a server

process for receiving RPC invocations. Atomic actions are employed to control the state changes to activated objects, and the properties of atomic actions given above ensure failure transparency. Atomic actions also ensure concurrency transparency, through concurrency control protocols, such as two-phase locking. Access transparency is normally provided by integrating an RPC pre-processor into the program development cycle which produces "stub" code for both the application and the object implementation. A variety of naming, binding and caching strategies are possible to achieve location and migration transparencies.

Normally, the persistent state of an object resides on a single node in one object store, however, the availability of an object can be increased by replicating it and thus storing it in more than one object store. Object replicas must be managed through appropriate replica-consistency protocols to ensure that the object copies remain mutually consistent. In a subsequent section we will describe how such protocols can be integrated within action based systems to provide replication transparency.

We assume some primitive features from a heterogeneous distributed system:

(i) The state of any object can have a context independent representation (i.e., free of references to a specific address-space). This implies that objects can be *de-activated* for storage or transmission over a network.

(ii) Executable versions of the methods of an object are available on all the nodes of interest. This implies that objects can be moved throughout the network simply by transmitting their states.

(iii) Machine-independent representations of data can be obtained for storage or transmission. This requirement is related to, but distinct from (i) in that this property enables interpretation of the passive state of an object in an heterogeneous environment.

Several prototype object-oriented systems have been built, often emphasising different facets of the overall functionality. For example, systems such as Argus [14], Arjuna [11, 23, 29], SOS [28] and Guide [4] have emphasised fault-tolerance and distribution aspects, languages such as PS-Algol [3], Galileo [2] and E [25] have contributed to our understanding of persistence as a language feature, while efforts such as [12] have contributed to the understanding of the design of object stores and their relationship to database systems. We build on these efforts and describe the necessary features of a modular distributed programming system supporting persistent objects.

3. System Structure

3.1. Computation model and System modules

With the above discussion in mind, we will first present a simple client-server based model for accessing and manipulating persistent objects and then identify the main system modules necessary for supporting the model. As stated earlier, we will consider an application program initiated on a single node to be the root of a computation; distributed execution is achieved by invoking operations on objects which may be remote from the invoker. We assume that for each persistent object there is at least 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 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 (do not survive the crash of the client or server). Bindings to servers are created as objects enter the scope in the application program. If some bound server subsequently crashes 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. An activated object which is no longer in use - because it is not within the scope of any client application - will not have any clients

bound to its server; this object can be de-activated simply by destroying the association between the object and the server process, and discarding the volatile image of the object (recall that the object will always have its latest committed state stored in some stable object store).

The disk 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 of an object will have an operation for acquiring, if necessary, a (read or write) lock from the associated 'lock manager' object 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 a distributed programming system, and the services they provide for supporting persistent objects.

- *Atomic Action module:* provides atomic action support to application programs in form of operations for starting, committing and aborting atomic actions;
- *RPC module:* provides facilities to clients for connecting (disconnecting) to object servers and invoking operations on objects;
- *Naming module:* provides a mapping from user-given names of objects to UIDs;
- *Binding module:* provides a mapping from UIDs to location information such as the identity of the host where the server for the object can be made available;
- *Persistent Object Support module:* provides object servers and access to stable storage for objects.

The relationship amongst these modules is depicted in Figure 1. Every node in the system will provide RPC and Atomic Action modules. Any node capable of providing object servers and/or (stable) object storage will in addition contain a Persistent Object Support module. A node containing an object store can provide object storage services via its Persistent Object Support module. Nodes without stable storage may access these services via their local RPC module. Naming and Binding modules

are not necessary on every node since their services can also be utilised through the services provided by the RPC module.

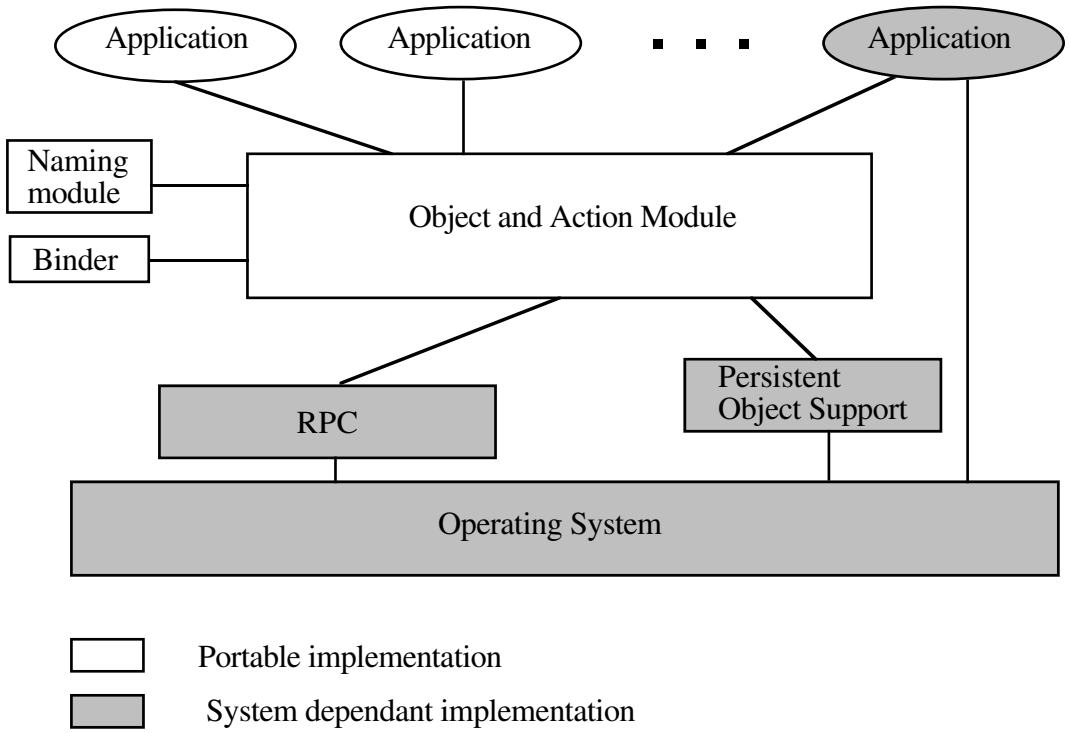

Figure 1: Components of a Persistent Object System

The above system structure also enables application programs to be made *portable.* An application program directly uses the services provided by the Atomic Action module which is responsible for controlling access to the rest of the modules. If all the persistent objects that an application references are accessed via the Atomic Action service interface, then the portability of the application depends only on the portability of the Atomic Action module implementation. The figure suggests that Atomic Action, Naming and Binding services can also be implemented in a system-independent, portable way. RPC and Persistent Object Support modules are necessarily system dependent at some level as they rely directly on operating system services. It is possible to make naming and binding services portable by structuring them as application level programs which make use of the Atomic Action module in a manner suggested above. The Atomic Action module itself can be made portable provided the services it requires from the RPC and Persistent Object Support modules are such that they can be easily mapped onto those already provided by the underlying system (for example, [6] describes how a uniform RPC system can be built by making use of existing RPC services). An application may well make use of the host operating system services directly (e.g., window management) in which case it can lose its portability

attribute (see Figure 1). Not surprisingly, the only way to regain portability is for the application to use portable sub-systems for all services (e.g., use the X Window System [26] for portable graphics services).

In the following discussion, we will initially make two simplifying assumptions: (i) an object can be activated only at the host node of the object store (that is, a node without an object store will not be able to provide object servers); and (ii) objects are not replicated. These restrictions will be removed subsequently.

3.2. Atomic Action Module

This module can be designed in two ways: (a) as a module providing language independent primitive operations, such as begin-*action, end-action* and *abort-action,* which can be used by arbitrary application programs; or (b) as an object-oriented, language specific run-time environment for atomic actions. The main advantage of the latter approach is that the ensuing class hierarchy provides scope for application specific enhancements, such as class-specific concurrency control, which are difficult to provide in the former approach. Although the choice is not central to the ideas being put forward here, we discuss the second approach (mainly because we have experience of building such an environment for C++ [30], as described in section 4).

To explain the functionality required from the Atomic Action module and the way it utilises the services of other modules, we will consider a simple C++ program (See Figure 2). In this simple example, an application program updates a remote persistent object, called `thisone`, which is of class `Example`, with the option of recovering the state of the object if some condition is not met. The application program creates an instance of an `AtomicAction`, called `A`, begins the action, operates on the object, then commits or aborts the action. We assume that this program will be first processed by a language specific *stub gen*erator (e.g., [23] for C++) whose function is to processes a user's application program to generate the necessary client-server code for accessing remote objects via RPCs. A detailed explanation of the steps follows:

```
1.  Example B ("thisone");  // bind to the server
2.  AtomicAction A;
3.  A.Begin();      // start of atomic action A
4.  B.op();         // invocation of operation, op on object B
5.  if (...) A.Abort();    // abortion of atomic action A
6.  else A.End();  // commitment of atomic action A
```

Figure 2: An atomic action example

Line 1: An instance, `B`, of (client stub) class `Example` is created by executing the constructor for that object. The string "`thisone`" is used at object creation time to indicate the name of the persistent object the program wants to access

(the identifier `B` acts as a local name for the persistent object `thisone`). As `B` is created, the following functions are performed (more precisely, these actions are performed by the client stub generated for `B`):

(i) an operation of the Naming service is invoked, passing the string "`thisone`" to obtain the UID of the object;

(ii) an operation of the Binding service is then invoked to obtain the name of the host (say α) where the server for the object can be made available; and finally,

(iii) an operation of the local RPC module is invoked to create a binding with the server associated with the object named by UID at node α; the binding is in the form of a communication identifier (CID), a port of the server, which is suitable for RPC communications. The details given in the descriptions of RPC and Persistent Object Support modules in the following subsections will make it clear how such a binding can be established.

Line 2: An instance, `A`, of class `AtomicAction` is created.

Line 3: A's `begin` operation is invoked to start the atomic action.

Line 4: Operation `op` of `B` is invoked by via the RPC module. As objects are responsible for controlling concurrency, the method of this operation will take any necessary steps, for example, acquiring an appropriate lock.

Line 5: The action may be aborted under program control, undoing all the changes to `B`.

Line 6: The `end` operation is responsible for committing the atomic action (typically using the two-phase commit protocol). This is done by invoking the `prepare` operation of the server of B (during phase one) to enable B to be made stable. If the `prepare` operation succeeds, the `commit` operation of the server is invoked for making the new state of the object stable, otherwise the `abort` operation is invoked causing the action to abort.

When `B` goes out of scope (this program fragment is not shown in the figure), it is destroyed by executing its destructor. As a part of this, the client-side destructor (the stub destructor for `B`) breaks the binding with the object server at the remote node (a specific RPC module operation will be required for this purpose). The functionality required from the RPC, Persistent Object Support, Naming and Binding modules can now be explained in more detail.

3.3. Remote Procedure Call Module

The RPC module provides distinct client and server interfaces with the following operations: `initiate`, `terminate` (which are operations for establishment and disestablishment of bindings with servers) and `call` (the operation that does the RPC), all these three operations are provided by the client interface, with `get_request` and `send_reply` being provided by the server interface. The operations provided by the RPC module are not generally used directly by the application program, but by the generated stubs for the client and server which are produced by a stub generator as mentioned before. Clients and servers have communication identifiers, CIDs (such as sockets in Unix), 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 implementation of `initiate(UID, hostname, ..)` operation involves the connection manager process co-operating with a local object store process (see the next subsection) to return the CID of the object server to the caller.

The client interface operations have 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 (normally this will occur because of the crash of the server and the client's response will be to abort the current atomic action, if any). The program structures shown in the previous sub-sections show that binding creation (destruction) can be performed from outside of application level atomic actions. So it is instructive to enquire what would happen in the presence of client or server failure before (after) an application level action has started (finished). The simple case is the crash of a server node, which has the automatic effect of breaking the connection with all of its clients: if a client subsequently enters an atomic action and invokes the server, the invocation will return exceptionally and the action will be aborted; 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 immediately after executing the statement in line 2 (figure 2). Then explicit steps must be taken to break the 'orphaned' binding: the server node must detect this crash and break the binding. The functionality of a connection manager process can be embellished to include periodic checking of connections with client nodes [22].

Every active object is associated with some object server; this server uses `get_request` and `send_reply` to service operation invocations. One server may manage several objects (i.e., the correlation between server processes and objects may

not be one-to-one). Any internal details of the server such as thread management for handling invocations are not relevant to this discussion.

3.4. Persistent Object Support Module

The Persistent Object Support module, with support from the RPC module, hides the (potential) remoteness of (stable) object storage systems from the applications; it also hides system specific details of stable storage, and provides a uniform service interface for persistent objects. This module is composed of two components: (i) an *object-manager* component, responsible for the provision of object servers; and (ii) an *object-store* component that acts as a front end to the local object storage sub-system. The object store representation (disk representation) of an object may differ from its volatile store representation (e.g., pointers may be represented as offsets or UIDs). We assume that the disk representation of objects are instances of the class `ObjectState`. Instances of class `ObjectState` are machine independent representations of the states of passive objects, convenient for transmission between volatile store and object store, and also via messages from node to node. A persistent object is assumed to be capable of converting its state into an `ObjectState` instance and converting a previously packed `ObjectState` instance into its instance variables (by using operations `save_state` and `restore_state` respectively). Figure 3 shows the state transformations of a persistent object along with the operations that produce the transformations (operations `read_state` and `write_state` are provided by the object-store component).

The primary function of an object-store component is to store and retrieve instances of the class `ObjectState`: the `read_state` operation returns the instance of `ObjectState` named by a UID and the `write_state` operation stores an instance of `ObjectState` in the object store under the given UID. In addition we assume two operations, `create` and `delete` for creating and deleting objects.

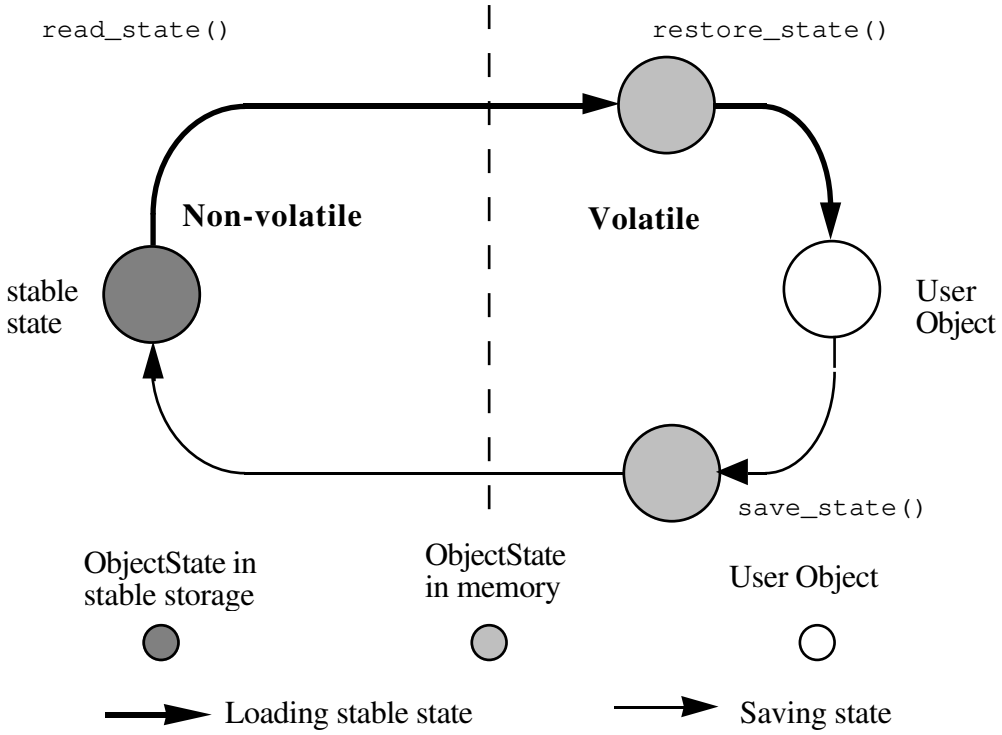

Figure 3: Object States

A typical implementation of the Persistent Object Support module would be as follows. The storage and retrieval of objects is managed by a *store demon* belonging to the object-store component. The sequence of events discussed previously with reference to the program fragment in Figure 2 can now be explained in terms of the activities at the Persistent Object Support module. Assume that the client program is executing at node N_1 and object `thisone` is at the object store of node N_2 (see Figure 4). The client process executing the program fragment will contain the stub for object B. Thus, at line 2, the client will execute the generated stub for B. This stub for B is responsible for accessing the naming and binding services as discussed earlier to obtain the location information for the object, and then to invoke the `initiate` operation of the local RPC module in order to send a connection request to the connection manager at N_2. Upon receiving such a request this manager invokes the `activate(UID)` operation provided by the object-manager. The object-manager is responsible for maintaining mappings between UIDs of activated objects to corresponding servers. Assume first that the object is currently active; then the object-manager will return, via the connection manager, the CID of the server to the client at N_1, thereby terminating the invocation of `initiate` at N_1. Assume now that the object is passive; then the object-manager will make use of some node specific activation policy based on which it will either create a new server for object B or instruct an existing server to activate B. That server uses the store demon for retrieving the `objectState` instance (by UID), loads the methods of B into the server, and invokes the `restore_state` operation of

B. The server acquires a CID and returns it to the client thus terminating the invocation of `initiate`.

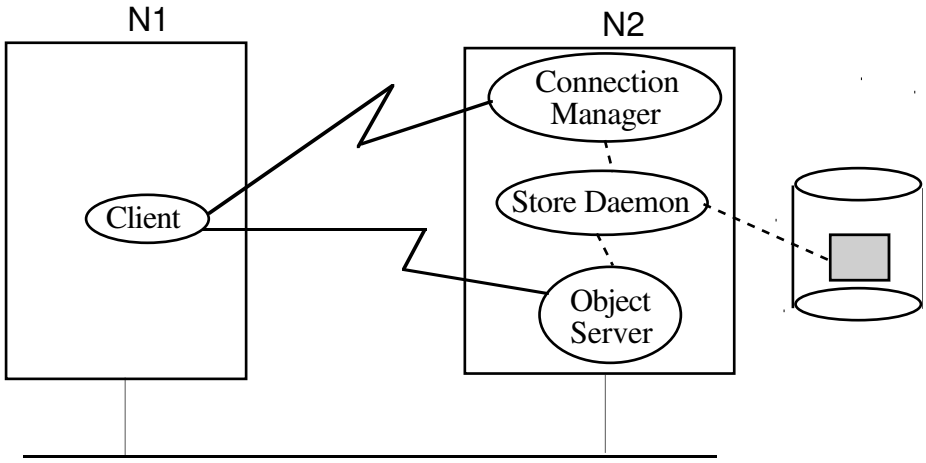

Figure 4: Accessing an Object

We introduce three additional operations of object-store component that are necessary for commit processing: `write_shadow`, `commit_shadow`, and `delete_shadow`. When the `prepare` operation for commit processing is received by the server, the volatile state of the object B will be converted into an instance of `ObjectState` (by using the `save_state` operation provided by B) and the object-store operation, `write_shadow`, will be invoked to create a (possibly temporary) stable version. If the server subsequently receives a `commit` invocation, it executes the `commit_shadow` operation of the object-store to make the temporary version the new stable state of the object. The response of the server to an `abort` operation is to execute the `delete_shadow` operation and to discard the volatile copy of the object.

To summarise: the Persistent Object Support module of a node provides eight operations: a single operation `activate` to the local connection manager process, and seven operations to local object server processes (`create`, `delete`, `read_state`, `write_state`, `write_shadow`, `delete_shadow`, `commit_shadow`); an object server itself provides operations `prepare`, `commit` and `abort` for commit/abort processing of the persistent object(s) it is managing. The operations a persistent object has to provide in order to make itself persistent and recoverable are `save_state and restore_state`.

It should be noted that an atomic action itself needs to record some recovery data on stable storage (e.g., an intentions list) for committing or aborting the action in the presence of failures. In the example considered, the intention list will be split between the client and server; however these details, which have been discussed

extensively in the literature, have been glossed over here. Support of nested and concurrent atomic actions further complicates the details of managing the commit records, but these aspects are also not central to the present discussion.

3.5. Naming and Binding Modules

The Naming and Binding services together support the location of objects by name and the management of naming contexts. Such services are often designed as a part of a single 'name server' which becomes responsible for mapping user supplied names of objects to their locations (e.g., [21]). However, these two services provide logically distinct functions related to applications. Whereas the object name to UID mappings maintained by the Naming module are expected to be static, the UID to location mappings maintained by the Binding module can change dynamically in a system supporting migration and replication.

The user-supplied names associated with objects are a convenience for the application programmer, not a fundamental part of the system's operation; within the system, an object is identified by its unique identifier, UID. The mapping from names of persistent objects to their corresponding UIDs is performed by the Naming service operation, `lookup`, which returns a UID. The Naming service itself can be implemented out of persistent objects by making use of the services provided by the Atomic Action module. This apparent recursion in design is easily broken by using well-known CIDs for accessing the Naming services. In addition to the `lookup` operation, the Naming service should also provide `add` and `delete` operations for inserting and removing string names in a given naming context. A naming service can always be designed to exploit an existing service (such as the Network Information Service [31]) rather than depending solely on the Atomic Action and other related modules for persistent object storage.

The Binding service, which maps UIDs to hosts, can also be designed as an application of the Atomic Action services. In addition to the `locate` operation, `add` and `delete` operations must also be made available. Enhancements to the functionality provided by the Binding service are required to support migration and replication of objects, as we discuss below.

3.6. Provision of Migration and Replication Transparencies

The architecture discussed so far possesses the functionality for supporting all the transparencies described earlier except replication and migration. We discuss now the enhancements necessary to support these two transparencies. First of all we observe

that the Naming service need not be affected, since it only maintains name to UID mappings for objects. The Binding service will be affected however, as for example a given object will be required to have its state stored on several object stores to support replication. This and other aspects are discussed below, starting with migration.

A simple but quite effective form of object migration facility can be made available by supporting migration during activation of an object: by permitting an object to be activated away from its object store node. This can be achieved by allowing the operations of a Persistent Object Support module to be invocable by remote object servers (and not just the local ones), thereby permitting an object server process to obtain the state (and methods) of the object from a remote object store. Thus a node without an object store can also now run object servers; such a node will contain a Persistent Object Support module, but without its object-store component. For the sake of simplicity, we will assume that the state and methods of an object are stored together in a single object store (this restriction can be removed easily without affecting the main ideas to be discussed below). One possible way of mechanising remote activation is discussed now. We assume that the object-manager component of a Persistent Object Support module now no longer maintains the mappings between UID to servers for activated objects, rather this information is made part of the Binding service. Thus, for a passive object, the `locate(UID)` function of the Binding service will return to the client the hostname of the object store node, together with a list of nodes where object servers can be made available, and for an active object, the pair (hostname, CID) indicating the CID of the object server managing the object at the node 'hostname'. A passive object will be activated as follows: from the list containing the names of potential server nodes and the object store node returned by the Binder, the client uses some criterion (e.g. the nearest node) for selecting a desirable server node for activation (say N_i), and then directs its `initiate` request to the connection manager process of N_i, giving it the name of the object store node (say N_k); at N_i an object server process gets the task of activating the object; this server fetches the necessary methods and state from N_k, acquires a CID and returns the CID to the client; the `initiate` operation terminates after the client has registered this CID with the Binding service. Registration with the Binder is necessary to ensure that any other client accessing this object also gets bound to the same server. Since we are assuming that an object is responsible for enforcing its own concurrency control policy - this to a large extent solves the problem of migrating concurrency control information with the object, since the "concurrency controller" of the object will move with the object. The scheme discussed here can be extended to permit movement of objects in between invocations, provided a client can locate the object that has since moved. A simple way of making migration information available to other clients is to leave a ,'forwarding address' at the old site so that any

invocations directed there can be automatically forwarded (see [8] for a more detailed discussion).

We turn our attention to the topic of replication transparency. We have so far assumed that the persistent state of an object resides on a single object store of a node; if that node is down, then the object becomes *unavailable*. The *availability* of an object may be increased by replicating it on several nodes and thus storing its state in more than one object store. Such object replicas must then be managed through appropriate replica-consistency protocols to ensure that the object copies remain mutually consistent. 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). We discuss below three aspects of replica consistency management; the first and the third are concerned mainly with the management of information about object replicas maintained by the Binding service, whereas the second is concerned mainly with the management of replicas once they have been activated.

(1) *Object binding:* It is necessary to ensure that, when an application program presents the name (UID) of an object which is currently passive to the Binding service, the service returns a list containing information about only those replicas of the object that are (a) mutually consistent, and also (b) contain the latest persistent state of the object. From this information, one, more, or all replicas, depending upon the replication policy in use (see below), can be activated. If the object has been activated already, then the Binding service must permit binding to all of the functioning servers that are managing replicas of the activated object. If we assume a dynamic system permitting changes to the degree of replication for an object (e.g., a new replica for an object can be added to the system), then it is important to ensure that such changes are reflected in the binding service without causing inconsistencies to the current clients of the object.

(2) *Object activation and access:* A passive object must be activated according to a given replication policy. We identify three basic object replication policies. (i) *Active replication:* In active replication, more than one copy of a passive object is activated on distinct nodes and all activated copies perform processing [27]. (ii) *Co-ordinator-cohort passive replication:* Here, as before, several copies of an object are activated; however only one replica, the *co-ordinator*, carries out processing [7]. The co-ordinator regularly checkpoints its state to the remaining replicas, the *cohorts*. If the failure of the co-ordinator is detected, then the cohorts elect one of themselves as the new co-ordinator to continue processing. (iii) *Single copy passive replication*: In contrast to the previous two schemes, only a single copy is activated; the activated copy regularly

checkpoints its state to the object stores where states are stored [5]. This checkpointing can be performed as a part of the commit processing of the atomic action, so if the activated copy fails, the application must abort the affected atomic action (restarting the action will result in a new copy being activated).

Activated copies of replicas (cases (i) and (ii)) must be treated as a single group by the application in a manner which preserves mutual consistency. Suppose the replication policy is active replication. Consider the following scenario (see figure 5), where group G_A (replicas A_1, A_2) is invoking a service operation on group G_B (a single object B) and B fails during delivery of the reply to G_A. Suppose that the reply message is received by A_1 but not by A_2, in which case the subsequent action taken by A_1 and A_2 can diverge. The problem is caused by the fact that the failure of B has been 'seen' by A_2 and not A_1. To avoid such problems, communication between replica groups can require reliable distribution and ordering guarantees not associated with non-replicated systems: reliability ensures that all correctly functioning members of a group receive messages intended for that group and ordering ensures that these messages are received in an identical order at each functioning member [27].

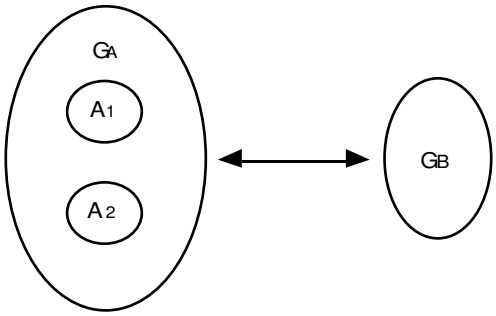

Figure 5: Operation Invocation for Replicated Objects

(3) *Commit processing*: Once an application has finished using an object, it is necessary to ensure that the new states of mutually consistent object replicas get recorded to their object stores; this takes place during the commit time of the application's atomic action. At the same time, it is also necessary to ensure that the information about object replicas maintained by the Binding service remains accurate. Consider an application that modifies some object, say A, and active replication is in use; suppose at the start of the application two replicas for A (A_1 and A_2) are available, but that the crash of a node makes one of them (say A_2) unavailable, so only A_1 gets modified; then at commit time, information maintained about A within the Binding service should be modified to 'exclude' A_2 from the list of available replicas of A (otherwise subsequent applications may end up using mutually inconsistent copies of A).

We conclude this subsection by observing that the introduction of migration and replication transparencies enforces consistency requirements on the Binding service that can be best met by composing the service out of persistent objects whose operations are structured as atomic actions (see [16] for more discussion).

4. Case Study: an Examination of Arjuna

We have arrived at the system structuring ideas presented in the previous section based on our experience of designing and implementing a distributed programming system called Arjuna [11, 23, 29]. Arjuna is an object-oriented programming system implemented in C++ that provides a set of tools for the construction of fault-tolerant distributed applications constructed according to the model discussed in section 2. Arjuna provides nested atomic actions for structuring application programs. Atomic actions control sequences of operations upon (local and remote) objects, which are instances of C++ classes. Operations upon remote objects are invoked through the use of remote procedure calls (RPCs). At the time of writing (December 1992), the prototype system has been operational for more than two years and has provided us with valuable insight into the design and development of such systems. The architecture presented in section 3 can be regarded as an idealised version of Arjuna.

4.1 Arjuna on Systems with Support for Networking Only

This section of the paper first describes the Arjuna system as designed and implemented to run on Unix workstations with just networking support for distributed computing (Unix sockets for message passing over the network); so all of the five modules shown in figure 1 (Atomic Action, Naming, Binding, Persistent Object Support and RPC modules) had to be implemented. In our discussion, we will be focusing on the approach taken to implementing the Atomic Action module.

The Atomic Action module has been implemented using a number of C++ classes which are organised in a class hierarchy that will be familiar to the developers of "traditional" (single node) centralised object-oriented systems. At the application level, objects are the only visible entities; the client and server processes that do the actual work are hidden. In Arjuna, server processes are created dynamically as RPCs are made to objects; these servers are created using the facilities provided by the underlying RPC subsystem, *Rajdoot*, also built by us [22]. The current implementation of Arjuna makes use of the Unix file system for long term storage of objects, with a class `ObjectStore` providing an object-oriented interface to the file system. The design and implementation of the Arjuna object store is discussed

elsewhere [11], along with the object naming (UID) scheme. This implementation strategy for the object store has been acceptable, but its performance is understandably poor. The naming and binding services themselves have been implemented out of Arjuna persistent objects.

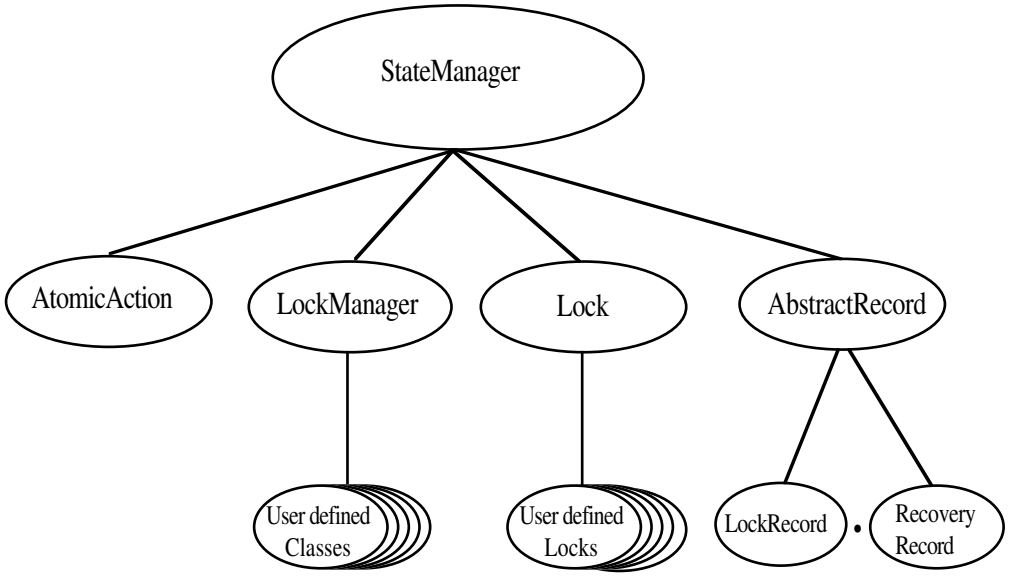

Figure 6: The Arjuna Class Hierarchy

The principal classes which make up the class hierarchy of Arjuna Atomic Action module are depicted in Figure 6. 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 in Figure 2; the operations this class provides (`Begin`, `Abort`, `End`) can then be used to structure 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 in Figure 6. Most Arjuna 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 mentioned before) and is responsible for performing aborts and commits.

Consider a simple example. Assume that O is a user-defined persistent object. An application containing an atomic action A accesses this object by invoking an operation op1 of O 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 appropriate operation of the concurrency controller (See Figure 7):

```
{
     // body of op1
     setlock (new Lock(WRITE));

     // actual state change operations follow
     ...
}
```

Figure 7: The use of Locks in Implementing Operations

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) use StateManager operations for creating a RecoveryRecord instance for O (the Lock is a WRITE lock so the state of the object must be retained before modification) and insert it into the RecordList of A;

(iii) create and insert a LockRecord instance in the RecordList of A.

Suppose that action A is aborted sometime after the lock has been acquired. Then the abort operation of AtomicAction will process the RecordList instance associated with A by invoking the abort operation on the various records. The implementation of this operation by the LockRecord class will release the WRITE lock while that of RecoveryRecord will restore the prior state of O.

The AbstractRecord based approach of managing object properties has proved to be extremely useful in Arjuna. Several uses are summarised here. RecoveryRecord supports state-based recovery, since its abort operation is responsible for restoring the prior state of the object. However, its recovery capability can be altered by refining the abort operation to take some alternative course of action, such as executing a compensating function. This is the principal means of implementing type-specific recovery for user-defined objects in Arjuna. The class LockRecord is a

good example of how recoverable locking is supported for a Lock object: the `abort` operation of `LockRecord` does not perform state restoration, but executes a `release_lock` operation. Note that locks are, not surprisingly, also treated as objects (instances of the class `Lock`), therefore they employ the same techniques for making themselves recoverable as any other object. Similarly, no special mechanism is required for aborting an action that has accessed remote objects. In this case, instances of `RpcCallRecord` are inserted into the `RecordList` instance of the atomic action as RPCs are made to the objects. Abortion of an action then involves invoking the `abort` operation of these `RpcCallRecord` instances which in turn send an "abort" RPC to the servers.

In the previous section we described three object replication approaches; out of these we have performed trial implementations of active and single copy passive replication in Arjuna [15, 17]. Active replication is often the preferred choice for supporting high availability of real-time services where masking of replica failures with minimum time penalty is considered highly desirable. Since every functioning member of a replica group performs processing, active replication of an object requires that all the functioning replicas of an object receive identical invocations in an identical order. Thus, active replication requires multicast communication support satisfying rigorous reliability and ordering requirements. Single copy passive replication on the other hand can be implemented without recourse to any complex multicast protocols (as only one replica carries out the computation at any time); however, its performance in the presence of primary failures can be poorer as it is necessary to abort the action and retry. We therefore believe that a fault tolerant system should be capable of supporting a number of replication schemes. The main elements of our design are summarised below.

(i) The Binding service (implemented as one or more Arjuna objects) maintains a 'group view database (GVD)' which records the information on available replicas of an object. The GVD itself can be replicated using either of the techniques to be described below. This database is accessed using atomic actions.

(ii) *Passive Replication:* To access an object (`A`), the application object first contacts the GVD, which returns a list containing location information on all the consistent replicas of `A`; a simple static ordering scheme is used for primary selection. The application object uses the RPC module operation `initiate(..)` for binding to the primary copy of `A`. If `A` itself accesses replicated objects, then the same technique is used again. At commit time, each primary object is responsible for updating its secondaries: this is made possible in Arjuna because the state of Arjuna objects can be transmitted over the network. If during the execution of an action, a primary is found to

become inaccessible (e.g., its node has crashed), then the action is aborted; as a part of the abort procedure, the GVD is accessed and the name of this primary is removed from the list of available replicas. Since actions can be nested, abortion need not be of the entire computation (the enclosing action can retry). When a crashed node containing a replica is repaired, it can include its copy of the object by running a *join* atomic action which updates the copy from some other replica and then inserts its name in the GVD's list for that object. In summary, the only major changes necessary to non-replicated version of Arjuna have been the creation and maintenance of GVD, and modifications to the abort and commit procedures as hinted above.

(iii) *Active Replication:* Activating an object now consists of activating all the copies listed in the group view list returned by the GVD. As atomic actions access replicated objects, a more accurate view of current group membership for an object is formed (if a copy is detected to have failed). At commit time, this current view is used for updating the GVD. Thus, any failed replicas automatically get excluded. The incorporation of active replication has meant the following two main changes to non-replicated version of Arjuna (in addition to the need for the creation and maintenance of GVD already discussed above):

- RPC module: the original unicast RPC has been replaced by a reliable group RPC, which is capable of invoking all the functioning copies of the object that were activated (in effect this has meant replacing the original datagram based RPC implementation by a reliable multicast protocol based one [15, 17]). In particular, the group RPC ensures that a replicated call from one group to another appears to behave like a single, non replicated call.

- Atomic Action module: the module is now responsible for manipulating object group view information. This means that an atomic action is required to maintain an 'exclude list' of replicas detected to have failed; at commit time this list is used for removing the names of these replicas from the group view list maintained by the GVD.

In summary, our approach has been to provide the basic binding information about object replicas via the GVD (an Arjuna object) which can then be used for providing either active or passive replication. The passive replication scheme has the advantage that it can be supported on top of any 'conventional' RPC system - this is important to a system like Arjuna which has been designed to be capable of exploiting the functionality offered by the underlying distributed system software.

The current design for Arjuna, while elegantly sorting out the functions of the Atomic Action module into classes, fails to separate the interfaces to the supporting

environment in the manner of section 3. The class `StateManager` combines operations relating to Persistent Object Support, RPC, Naming and Binding. The management of recovery, persistence, distribution and concurrency control is well-organised around the classes discussed before, but the interfaces to the services are not so well organised. The present RPC facility, while supporting the interface discussed, is also responsible for the creation of object servers, a function which should be performed by the Persistent Object Support module. The Naming and Binding services have not been properly separated, their combined functions currently being performed by a simple name server. Revisions to the system to carry through the object-oriented design along the lines presented in Section 3 of this paper are currently underway. These revisions however do not represent a major overhaul of the system. Thus the system demonstrates that distributed systems structured along the lines of Figure 1 can be built.

4.2. Arjuna on other systems

We will now describe how the Arjuna system described above has been adapted to run on two quite different systems providing basic support for distributed computing (e.g., RPC), enabling the Atomic Action module of Arjuna to utilise some of the services of the host system in place of the services of the modules built earlier. It is because our system has the modular structure proposed here that we have been able to perform such ports.

Our first port has been on to the ANSAware distributed computing platform. The ANSAware platform has been developed by the ANSA project [1]; the platform provides RPC, object servers (known as *capsules*) and naming and binding services via a subsystem known as the *Trader*, for networked workstations (several operating systems are supported; we have so far used only Unix). This porting has been a relatively straight forward exercise. To start with, we have removed the RPC module used in the original Arjuna (Rajdoot) and mapped the RPC operations (`initiate`, `terminate` and `call`) onto those provided by ANSAware. This enables Arjuna applications to run on top of ANSAware; this port automatically supports passive replication. In the near future we will enhance this port to use the ANSAware Trader for registering Arjuna naming and binding services. The ANSAware system has recently been upgraded to support group invocations [20] for active replication and object storage services [19]. We believe that these services can also be used in place of the original Arjuna services used for supporting active replication and object storage.

We have also performed experiments to ascertain whether Arjuna can be made to run on integrated environments provided by distributed operating systems [10]. The

experimental configuration we have used consists of a locally distributed multiprocessor system of twelve T800 transputers, each with 2 Mbytes of memory, interconnected to form of a two-dimensional grid (see figure 7). Each transputer runs a copy of Helios, which is a is a general-purpose distributed operating system [24]. The Helios file server program (hfs) running on one of the transputers provides access to a disk, which is used as an object repository.

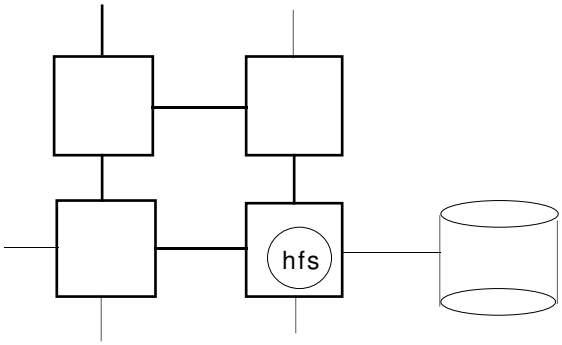

Figure 8: A multi-transputer system

The Helios operating system provides a number of facilities for client-server programming. Helios treats every file, process, and device, including processors, as an object, each of which can be named using Unix like path names. Each object is represented by an *Object-structure* which contains information such as the full pathname of the object, and the object type e.g. file, process etc. The Helios *Locate* function allows an Object-structure to be obtained for any object in the system, given its name. The function accesses a local (to a processor) name server which can initiate a flood search throughout the system if the Object-structure is not available locally. As a result of the search the local name server is updated with the relevant Object-structure and subsequent locates for that object are handled entirely locally. Once an object has been located it may be opened through the use of the Helios *Open* function. If the object is a process then the Object-structure contains the Helios port via which messages may be sent to that process using the Helios *PutMsg* function. Messages are received on a port using the Helios *GetMsg* function. A process can act as a server by binding one of its communications identifier, CID, to a name (a *service name*), registering that service name with the local Helios name server and waiting for communication over that CID. Any process may obtain the CID of a registered server by using the Locate function.

To port Arjuna, we have implemented a number of Helios application programs, collectively known as the *object management layer*. This layer implements an RPC facility using PutMsg, GetMsg functions of Helios, and object servers which are mapped onto a Helios servers which may then register themselves as discussed above. Such a server may then receive Open requests from clients on the communication port associated with the service name. Although several shortcuts have

been taken in this exercise (e.g. client and server stubs have been hand crafted), the experiment does show that the functionality required by Arjuna Atomic Action module can be mapped, via the object management layer, onto the underlying services provided by Helios.

6. Concluding Remarks

This paper has presented a modular architecture for structuring fault-tolerant distributed applications. 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 atomic action based object-oriented systems. We have arrived at the ideas presented here based on our experience of building the Arjuna system which can be made to run on a number of distributed computing platforms.

The Atomic Action module provides a fixed means of combining the above stated object properties. We are now investigating whether these can be provided individually, permitting application specific selection. Such a system for example could permit shareable objects that need not be persistent, and vice-versa (although they could be). Furthermore these properties can be enabled and disabled at run-time based on application requirements. Our initial work in this direction reported in [9, 18] indicates that this is indeed possible.

Acknowledgements

The Arjuna project has been and continues to be a team effort. Critical comments from Graham Parrington, Mark Little and Stuart Wheater are gratefully acknowledged. Continued interactions with colleagues on the ANSA-ISA project have proved beneficial. The ANSAware and Helios ports have been performed by Joao Geada, Stuart Wheater, Jim Smith and Steve Caughey.

References

[1] Advanced Networked Systems Architecture (ANSA) Reference Manual, Volume A, Release 1.00, Part VI, Computational Projection, March 1989, (available from APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD, UK).

[2] Albano, A., Ghelli, G., and Orsini, R., "The Implementation of Galileo's Persistent Values", in Data Types and Persistence, Atkinson, M.P., Buneman, P., and Morrison, R. (eds.) , Springer-Verlag, 1988, pp. 253-263.

[3] Atkinson, M.P., K. J. Chisholm and W. P. Cockshott, "PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices, 17, 7, July 1981, pp. 24-31.

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

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

[6] Bershad, B.N., et al, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems", IEEE Transactions on Software Engineering, Vol. SE-13, No. 8, pp. 880-894, August 1987.

[7] Birman, K. and T. Joseph, "Exploiting virtual synchrony in distributed systems", in 11th Symposium on Operating System Principles, ACM SIGOPS, November 1987.

[8] Black, A. and Y. Artsy, "Implementing Location Independent Invocation", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, pp. 107-119, January 1990.

[9] Caughey, S.J., G.D. Parrington and S.K. Shrivastava, "Shadows: a flexible run-time support systems for objects in a distributed system", Technical Report, Department of Computing Science, University of Newcastle upon Tyne, December 1992.

[10] Caughey, S.J., S.K. Shrivastava and D.L. McCue, "Implementing fault-tolerant object systems on distributed memory multiprocessors", Proc. of 2nd IEEE Intl. Workshop on Object Orientation in Operating Systems, pp. 172-179, Dourdan, France, September 1992.

[11] Dixon, G.N, et al, "The Treatment of Persistent Objects in Arjuna," Proceedings of the Third European Conference on Object-Oriented Programming, ECOOP89, pp. 169-189, July 1989. (Also, The Computer Journal, Vol. 32, No. 4, pp. 323-332, 1989).

[12] Garza, J. and W. Kim, "Transaction Management in an Object-Oriented Database System", Proceedings of ACM SIGMOD Conference on Management of Data, pp. 37-45, Chicago, Sept. 1988.

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

[14] Liskov, B., and R. W. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, pp. 381-404, July 1983.

[15] Little, M., and S.K. Shrivastava, "Replicated K resilient objects in Arjuna" Proc. of IEEE Workshop on the Management of Replicated Data, Huston, Texas, November 1990, pp. 53-58.

[16] Little, M.C., D.L. McCue and S.K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", Technical Report, Department of Computing Science, University of Newcastle upon Tyne, October 1992.

[17] Little, M.C., "Object Replication in a Distributed System", PhD Thesis, Department of Computing Science, University of Newcastle upon Tyne, September 1991.

[18] McCue, D.L., "Developing a class hierarchy for object-oriented transaction processing", Proc. of European Conference on Object-Oriented Programming, ECOOP 92, pp. 413-426, Utrecht, June 1992.

[19] Olsen, M.H., "A persistent object infrastructure for heterogeneous distributed systems" Proc. of 2nd IEEE Intl. Workshop on Object Orientation in Operating Systems, pp. 49-56, Dourdan, France, September 1992.

[20] Olsen, M.H., E. Oskiewicz and J.P. Warne, "A model for interface groups", Proc. of 10th IEEE Symposium on Reliable Distributed Systems, SRDS-10, Pisa, October 1991.

[21] Oppen, D.C., and Y.K. Dalal, "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment", Technical Report of Xerox Office Products Division, Palo Alto CA, OPD-t8103, October 1981.

[22] 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.

[23] Parrington, G.D. "Reliable Distributed Programming in C++: The Arjuna Approach", Proceedings of the USENIX Second C++ Conference, San Francisco, pp. 37-50, April 1990.

[24] Perihelion Software Ltd.,"The Helios Operating System", Prentice Hall International, ISBN 0-13-386004-3, 1989.

[25] Richardson, J.E., and M.J.Carey, "Persistence in the E Language: Issues and Implementation", Software Practice and Experience, 19 (12), pp. 1115-1150, Dec. 1989.

[26] Scheifler, R.W., and J. Gettys, "The X Window System", ACM Transactions on Graphics, Vol 5, No. 2, pp. 79-109, April 1986.

[27] Schneider, F.B., "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, December 1990.

[28] Shapiro, M., P. Gautron and L. Mosseri, "Persistence and migration for C++ objects", Proceedings of the Third European Conference on Object-Oriented Programming, ECOOP 89, Nottingham, pp. 191-204, CUP, ISBN 0 521 38232 7, July 1989.

[29] Shrivastava, S.K., G. N. Dixon and G.D. Parrington, "An overview of the Arjuna distributed programming system", IEEE Software, pp. 66-73, January, 1991.

[30] Stroustrup, B., The C++ Programming Language, Addison Wesley, 1986.

[31] Sun Microsystems, "Network Programming", Mountain View, CA, Revision A, May 1988.