COMPUTING
SCIENCE

Implementing Fault-tolerant Object Systems on Distributed Memory Multiprocessors

S.J. Caughey, S.K. Shrivastava and D. McCue

TECHNICAL REPORT SERIES

No. 391 September, 1992
Implementing Fault-tolerant Object Systems on Distributed Memory Multiprocessors

S.J. Caughey, S.K. Shrivastava and D. McCue

Abstract

The design and implementation of an object management layer for a distributed memory multiprocessor system is described. It has been used for supporting an object-oriented fault-tolerant system (Arjuna). We discuss how various aspects of distribution transparency (location, access, migration, concurrency, replication and failure) have been incorporated in our design.
Bibliographical details

CAUGHEY, Steven John
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no.391)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE,
SHRIVASTAVA, Santosh Kumar
MACCUE, Daniel Lawrence

Abstract

The design and implementation of an object management layer for a distributed memory multiprocessor system is described. It has been used for supporting an object-oriented fault-tolerant system (Arjuna). We discuss how various aspects of distribution transparency (location, access, migration, concurrency, replication and failure) have been incorporated in our design.

About the author

Steve Caughey joined the Department of Computing Science in October 1989 after he completed his CSSD course. He is a research associate currently working on fault tolerance for distributed memory multiprocessors.

S.K. Shrivastava joined the Department of Computing Science in August 1975, where he is a Professor.

Dan McCue is a member of the Arjuna project working on reliable distributed systems research. He has focused on the design of object-oriented, distributed transaction systems, especially issues of integration of distribution and transaction systems concepts into object-oriented languages.

Suggested keywords

ATOMIC ACTIONS FAULT TOLERANCE HELIOS
OBJECT SYSTEMS PERSISTENCE TRANSPUTERS

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425
U.D.C. 519.687 681.322.06
Implementing Fault-tolerant Object Systems on Distributed Memory Multiprocessors

S.J. Caughey, S.K. Shrivastava and D. McCue

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

Abstract

The design and implementation of an object management layer for a distributed memory multiprocessor system is described. It has been used for supporting an object-oriented fault-tolerant system (Arjuna). We discuss how various aspects of distribution transparency (location, access, migration, concurrency, replication and failure) have been incorporated in our design.

1. Introduction

Distributed memory multiprocessor systems offer scope for constructing processing systems which can scale easily to accommodate large numbers (from hundreds to thousands) of processors. A particularly promising underlying architecture is that of individual computers (each comprising a processor plus memory) interconnected by (redundant) crossbar switches. Computer systems based on such an architecture can be useful in a variety of application domains. For example, future distributed systems are expected to rely increasingly on such multiprocessor systems for providing high performance, shared and available object storage services to clients over the network. A multiprocessor node providing storage services for user specific persistent objects can employ a number of discs to implement a parallel storage system for obtaining large storage capacity and improved performance [1]. To take another example, a multimedia workstation could be built by employing a set of disk, video, audio etc. devices, (each with its own processing element) and computing nodes, connected by a high-speed switch, with each device running dedicated specialised servers. In the domain of telecommunications, a switching system could be built out of set of devices (each with its own processing element) providing access to subscriber or trunk communications systems, and computing nodes, connected by a high-speed switch.

With the possibility of large numbers of processors within such 'locally distributed' nodes, the issue of fault-tolerance becomes important. We are interested in developing a software systems architecture for such nodes capable of providing fault-tolerant services to clients, where a client could be executing either on the node or remote from the node. To this end we have adapted an object-oriented fault-tolerant system - Arjuna - developed for workstation-based distributed systems [2,3] to run on a locally distributed multiprocessor system. In this paper we describe the architecture of our system and a preliminary implementation. Our approach has been to implement a general purpose object management layer which exploits the specific features of the underlying operating system - in our case this is the Helios distributed operating system [4] running on multiple transputers - and then to use this layer to support the functionality required by a specific system, Arjuna (in this respect our object management layer for Helios performs broadly similar functions to those for say, Chorus and SOS [5,6]). Given that we want to support distributed fault-tolerant computations, an important design decision concerns determining how the overall functionality required for supporting distribution and fault-tolerance be split between the lower object management layer and the upper layer. Our views on the above subject have been strongly influenced by the experience of building the Arjuna distributed system which provides facilities for persistent objects (instances of C++ classes in the current implementation) which can be manipulated under the control of atomic actions (atomic transactions). Fault-tolerance is necessary primarily for preserving the integrity of persistent objects - that is why atomic actions are required. The Arjuna system has been designed to be portable [7]; as such it has been structured to require only the minimum support for fault-tolerance, persistence and distribution from the underlying layer. This portable structure simplifies the design and implementation of the object management layer on Helios.
Helios is a general-purpose operating system for parallel computers, originally designed for transputers. Helios is based on the client-server model and provides the ability for servers to register themselves with a binder, enabling clients to locate named servers, and for clients and servers to communicate using messages, independent of physical location. The experimental configuration we use comprises a node consisting of twelve T800 transputers, each with 2 Mbytes of memory, interconnected to form a two-dimensional grid. Each transputer runs a copy of Helios, and the whole setup is connected to a Unix workstation whose filing system can also be used by Helios. Although Helios is not promoted as an object-oriented system it does provide features that ease the production of such a system. In particular, Helios treats every file, process, and device, including processors, as an object, and access to these objects is achieved in an almost identical manner independent of the object type. We will describe how these features have helped us in our design.

We begin by describing the model of fault-tolerant computation; this will help us to explain the functionality required from the object management layer for Helios.

2. Model of computation

We will first present the essential aspects of an object-oriented model of computation. We will not make any assumptions about the underlying architecture, and simply assume that objects are capable of communicating with each other by sending messages. To develop our ideas, we will first describe some desirable transparency properties a distributed system should support. There are several complementary aspects to transparency [8]:

- **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;
- **Concurrence transparency** mechanisms ensure interference-free access to 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.

The computational model we use to incorporate these forms of transparencies is based upon the well-known concept of using atomic actions (atomic transactions) controlling operations on persistent 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. An operation invocation upon a remote object is performed via a remote procedure call (RPC), which hides the details of request-reply message interactions. Furthermore, 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.

The object and atomic action model provides a natural framework for providing fault-tolerant services using persistent objects. In this model, persistent objects not in use are held in a passive state, residing in object stores or object databases and activated on demand. Atomic actions (initiated from clients) are employed to control the state changes to these 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. An object resides in one object store, however, the availability of an object can be increased by replicating it and 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. Such protocols can be integrated within action based systems to provide replication transparency.

From the preceding discussion, the essential components of a persistent object system can be identified:

1. Atomic Action services: provide atomic action support to application programs;
2. RPC services: provide an object invocation facility through an RPC mechanism;
(3) Object Storage services: provide a stable storage repository for objects; these object are assigned unique identifiers (UIDs) for naming them;
(4) Naming services: provide a mapping from user-given names of objects to UIDs;
(5) Binding services: provide a mapping from UIDs to location information i.e., the identity of the server for the object.

Persistent objects are normally resident on object stores (such an object is said to be in a passive state). An active copy of a passive object is made by loading its state and methods from the object store to the volatile store. Note that this operational description of the activation and deactivation of objects is intended to convey the semantics of the operations; an implementation of the object store is free to use caching, pre-fetching, clustering, etc. to optimise the movement of objects (code and data) from stable storage to volatile storage and back.

To illustrate the interactions between the Atomic Action and other modules, we will consider a simple example (taken from Arjuna). In this example, an application programmer wants to update a persistent object of the class Example with the option of recovering the state of the object if some condition is not met. The application program creates an Atomic Action, called A, begins the action, operates on the object, then commits or aborts the action. We assume that this program has been processed by a language specific stub generator (eg., [9] 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 (in effect, on the client's side, object B will be replaced by a stub object, which is responsible for making RPCs to the real object held at the server side). A detailed explanation of the steps follows:

1. AtomicAction A;
2. Example B ("thisone");
3. A.Begin();
4. B.op();
5. if (...) A.Abort();
6. else A.End();

**Fig.1: An atomic action example**

Line 1: An instance, A, of class AtomicAction is created.
Line 2: An instance, B, of class Example is created. The string "thisone" is used at object creation time to access the persistent object by that name (the name "B" acts as a local name for the persistent object named "thisone"). The client stub constructor for B performs the following the following steps:
(a) the lookup operation of the Naming service is invoked, passing the string "thisone" to obtain the UID of the object;
(b) the locate operation of the Binding service is invoked to obtain the name of the server who can be relied upon to fetch the object state; and finally,
(c) this server is called upon to activate the object, and return a communication identifier (CID) suitable for RPC communications. Once an object has been activated, its operations can be invoked via the RPC module, using the CID associated with the activated object.

Line 3: A's begin operation is invoked to start the atomic action.
Line 4: The operation B.op(...) will succeed if there are no failures and there are no conflicting operations on B already in progress. If the invocation is refused due to a conflict, it could be re-tried some number of times, blocked until the object becomes available, or the entire action, A, could be aborted and re-tried.
Line 5: The action may be aborted under program control, undoing all the changes to B.
Line 6: The end operation commits the atomic action (using the two-phase commit protocol). This is done by the invoking the prepare operation of B (during phase one) to enable B to be made stable. If the prepare succeeds, the commit operation of B is invoked (during phase two) otherwise the abort operation of B is invoked as the action aborts. These three operations can be provided by B through inheritance from classes of the Atomic Action services.

3. Designing the Object Management Layer (OML)

From now on we will concentrate on the various aspects of our system architecture pertaining to a single (locally distributed) node, and for the sake of simplicity assume that clients are accessing objects residing on the same node (it is relatively easy to arrange for clients to access objects available on other nodes). We will make the following assumption regarding application level objects:

Objects are responsible for providing the necessary operations required for persistence, commit, and abort and are capable of performing their own concurrency control. For persistence, it is necessary for an object to be able store its current state onto the disk. The disk representation of an object may differ from its volatile store representation (e.g., pointers may be represented as offsets or uids). The disk representations of objects will be assumed to be 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 processor to processor. A persistent object is assumed to be capable of packing its state into an ObjectState instance and unpacking a previously packed ObjectState instance into its instance variables by invoking operations on ObjectState objects. This packing and unpacking capability is sufficient for supporting recoverability, as it enables an object to checkpoint its state and restore a prior state from a checkpoint. The example in the previous section indicated how commit, abort and concurrency control operations of an object are utilised from within atomic actions; in a subsequent section we will briefly describe how the Arjuna system provides a number of classes which enable a user level object to inherit much of the functionality implied here.

Given this assumption, the main role of the OML reduces to: (i) providing an RPC mechanism that allows clients to invoke operations on objects; and, (2) providing a means for activating and passivating objects: the former entails locating the passive object in (stable) object store and loading the object’s state and methods into a server process running on some processor of the node, returning a communications identifier (CID), a message port number of the server, to the client who can then invoke operations on that object; the latter entails storing the current state of the object back on the object store. It is only necessary for the OML to manage two kinds of objects: of class ObjectState (instances of which encode states of objects), and of class ObjectMethod (instances of which are object methods).

OML provides the facility for an object in the system to be accessed in either local or global mode. An object which has been made passive but which is to be accessed must be co-located with the methods for that object’s class. If the access is to be local only then any process which has linked in the appropriate methods can request the ObjectState instance from OML. If the access is to be global then the access is performed via a globally accessible object server. Each such global object is managed by a server process (see below) which accepts RPC requests for access to a particular object, carries out the access and returns the result. An RPC system has been implemented using the Helios client-server messaging facility (the general server protocol [4]). It provides three operations to a client: initiate (...) for opening a connection to an object whose name (which could in the form of a string) is supplied as a parameter; initiate returns a CID to the caller who can then use this for making RPCs, using a call(...) operation; the third operation is terminate(...) for breaking a connection.

The central concept of the OML design is the provision of ObjectServers, multi-threaded server processes, each of which manages objects of a particular type; OML is responsible for creation and deletion of ObjectServers. An ObjectServer for objects of say, class X, will contain the methods of X and instance variables of objects which have been activated. An ObjectServer for class X has the ability to accept an ObjectState instance (representing the packed state of an object of class X), which can then be unpacked to initialize instance variables, and to give away an ObjectState instance (representing the packed state of one of the objects of class X the server is managing) if requested. By this means objects may be migrated around the system. In summary, an ObjectServer exports three operations: accept(...), give(...) and execute(...); the latter operation is for executing a given method, whose name (an opcode) is supplied as a parameter.

The structure of an ObjectServer is depicted in fig. 2. The fig. shows the server for object type X, managing three instances, x1, x2 and x3, out of which x1 and x2 have their states loaded from the object store but the state of x3 has not yet been loaded. An object table within the server is used for locating the state (instance variables) of an object, given its UID. An object server has a dispatcher thread listening on a CID whose name has been published (thereby made known to potential clients). Upon receiving a method invocation for a particular object from a new client, this thread will fork a new thread to serve this and subsequent invocations from this client (a new CID, such as CIDj, is acquired for receiving subsequent invocations from the client); the thread uses the object table for locating the state of the object in question. If the state has not been loaded, then the state must be retrieved from the object store.

Fig. 2: An ObjectServer

OML permanently runs a few ObjectServers which are necessary for the management of user level objects:
(i) **ObjectState Servers**: these servers are responsible for interfacing to object stores; an ObjectState Server can store and retrieve instances of ObjectState objects, given their UIDs. It also supports operations necessary for commit processing: write_shadow, make_permanent, and delete_shadow. ObjectState servers are intended to act as front ends to stable object store repositories available to the system (these could be either local or remote). When the prepare operation is received by the ObjectServer managing some object, say B, it will pack the volatile state of the object B into an ObjectState instance hand it over to the relevant ObjectState server by invoking its execute(.write_shadow...) operation, which will have the effect of creating a (possibly temporary) stable version. If the server for B subsequently receives a commit invocation, it will invoke execute(.make_permanent...) operation of the ObjectState server to make the temporary version the new stable state of the object. The response of the server for B to an abort operation is to execute the delete_shadow operation in order to discard the shadow version.

(ii) **ObjectMethod servers**: such a server is capable of storing and retrieving the code for objects (which are instances of ObjectMethod) to and from some object store. The code for an object will be required only when it is necessary to start up a server for that object type; so an ObjectMethod server takes on the responsibility for creating an ObjectServer for a given class and loading the code for the class in it. The server can potentially be created on any of the functioning processors (ObjectMethod servers can be designed to employ some specific criterion, eg., chose a lightly loaded processor, for server creation).

(iii) **NameServer**: maintains a mapping from user-given names of persistent objects to UIDs. An object has actually two UIDs associated with it: one for the ObjectState and other for the ObjectMethod (referred respectively as UIDOS and UIDOM)

(iv) **Binder**: maintains the following mappings for each object:
   (a) UIDOS to the name of the ObjectServer which is serving objects of that type (if such a server exists);
   (b) UIDOS to the name of the StateObject server from which to fetch the state of the object;
   (c) UIDOM to the name of the ObjectMethod server from which to fetch the methods of the object.

We can now describe how a client program can obtain a connection to a named object by invoking the initiate operation. The initiate operation basically performs the three steps described in connection with line 2 of the example in the previous section (see fig. 1). A simple case would be when a client is accessing an object which is already active, in which case the binder will return the name of the relevant ObjectServer. We discuss the case when it is necessary to create an ObjectServer, as such a server for the named object (say x1 of class X) does not exist (the binder has no UIDOS to the server name mapping) in which case the binder returns the name of the ObjectMethod server; initiate then contacts that ObjectMethod server which creates an ObjectServer (as discussed earlier) loaded with the methods of X, and returns the name for this server. The client can now contact this server; upon receiving an invocation, and finding that the state of x1 has not yet been loaded, the ObjectServer can fetch the state by first contacting the binder to receive the name of the ObjectState server and then that server for the ObjectState instance, and then executing the unpack method of X, to initialize x1 with this state.

4. Implementation and assessment

4.1. Implementation

We have implemented a prototype of OML and have used it to support the Arjuna system. A few shortcuts have been made in this port (eg., we have not yet implemented StateObject and MethodObject servers but have made direct use of the Helios file system, and client and server stubs have been hand crafted), but the port demonstrates that OML has the desired functionality. The current hardware configuration consists of twelve T800 transputers, each with 2 Mbytes of memory, interconnected to form a grid; each transputer runs the Helios operating system. The Helios file server program (hfs) running on one of the transputers provides access to a disk, which is used as an object repository. The type of architecture we see as the eventual target for our system would consist of individual computers (perhaps T9000 transputers) interconnected by (redundant) crossbar switches (eg., C104 for T9000).

![Fig. 3: A multi-transputer system](image-url)
The Helios operating system provides a number of interesting features including cheap processes and threads (on a transputer), and the provision 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 initiates 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.

Helios provides a Resource Management Library which enables a user to examine the current network, in terms of its constituent processors and the links between those processors. This information may be used by system programs (such as OML) to make decisions regarding the optimum placement of servers and the desirability of individual object migration. Processors may be obtained from the network and may optionally be designated as being private i.e. unavailable to other users. By this means processors may be reserved for performance critical servers or user programs protected from interference from other users. Processors may be dynamically obtained and released as required. Additionally, the library provides a simple mechanism which allows system programs to initiate servers anywhere within the network (providing the chosen processor has not been reserved by another user). It also provides a function which allows any process to execute a program transparently on any processor by specifying in effect the processor name and the name of the executable file. Similarly, task forces (a set of communicating, cooperating processes e.g. a replication group) may be created and mapped onto the network.

A process can act as a server by binding one of its CIDs 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.

From the above discussion, it should be clear that an ObjectServer can be mapped onto a Helios server which may register itself as discussed above. It may then receive Open requests from clients on the communication port associated with the service name. Open requests are processed by a dispatcher thread which creates a new thread which is then responsible for the client-server interactions, as indicated in fig. 2. A number of C++ classes have been written to facilitate the construction of such servers [10]; these classes have also been used for the implementation of an RPC system.

Name server and binder objects have been implemented as persistent user defined Arjuna objects: so, operations on these objects can be performed atomically, if desired. A user defined Arjuna class can inherit the properties of persistency, recovery and concurrency control. We briefly describe how this is done.

4.2. The Arjuna class hierarchy

The principal classes which make up the class hierarchy of Arjuna are depicted in Fig. 4. These classes represent the internal structure of the Atomic Action module. 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 Fig. 1; 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 Fig. 4. All Arjuna classes are derived from the base class StateManager, which provides the primitive facilities necessary for constructing persistent objects and atomic actions. These facilities include support for the activation and de-activation of objects, and object recovery (operations for packing and unpacking are provided as virtual functions, whose code is provided by users).

![Fig. 4: The Arjuna Class Hierarchy](image_url)

The class LockManager uses the facilities of StateManager and provides the concurrency control (two-
phase locking in the current implementation). The implementation of atomic actions, recovery, persistence management and concurrency control is organized as a collection of object classes derived from the class, AbstractRecord which is in turn derived fromStateManager. 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. In summary, to construct a persistent object, a user basically has to derive the class fromLockManager, and provide routines for packing instance variables to an instance of ObjectState and for unpacking an instance of ObjectState to the instance variables; further, methods must be programmed to contain read or write lock operations. This is straightforward, as the following example illustrates (for some operation, op1 of class O, which requires a write lock, as the operation modifies the instance variables):

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

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

**Fig. 5 Setting a lock**

### 4.3 Assessment

The approach we have followed in our design has been to push much of the functionality required for fault-tolerance and distribution to application level. The object management layer is then concerned with the provision of generic object servers, which themselves rely on the underlying operating system for communication and resource management. Location transparency has been handled at the application level by maintaining naming and binding information for user objects in servers which can be (and have been) programmed as application level objects, and then relying on the operating system location service for locating a server given its name. Concurrency control is handled entirely at the application level. Access transparency has been handled at the operating system level (within OML), while failure transparency requirements have been handled at the application level given the availability of stable storage. Implementing replication and migration transparencies on the other hand requires a subtle interplay between OML and the application level as we discuss below.

The version of Arjuna which has been ported on Helios does not support replication of objects. We have designed and implemented replication schemes for Arjuna, but these have not yet been fully integrated into the Helios version. Nevertheless, we can make some useful observations regarding the support required from OML. There are basically two types of replication techniques: active and passive. In active replication each and every correctly functioning member of a replica group performs processing, whereas in passive replication only one member of the group, the coordinator (primary), performs the processing and checkpoints its state to the rest of the secondary replicas. 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. We believe that a system should be capable of supporting both types of replication schemes.

The active replication scheme for Arjuna is described in [11]. The incorporation of active replication will essentially require the following enhancements:

(i) The original unicast RPC has to be replaced by a reliable group RPC, which is capable of invoking all the functioning copies of the object (in effect this means replacing the original datagram based RPC implementation by a reliable multicast protocol based one [11]). In particular, the group RPC should ensure that a replicated call from one group to another appears to behave like a single, non replicated call.

(ii) 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 name server.

(iii) Naming service: the service is now required to maintain group view information about replicated objects.

The last two modifications do not impose any new requirements on OML, but the group RPC does, in that a reliable multicast protocol is required. Such a protocol can be built on top of the existing datagram service. Turning our attention to passive replication, it is possible to design a scheme that can be supported on top of the current OML: basically at commit time, all the replicas on functioning processors are updated; if the primary fails during a computation, the action is aborted (and later retried with one of the secondaries as the new primary). Thus, no
additional functionality is required from OML but this is at expense of the cost of action aborts.

Object migration can be handled, within the framework described here by the following procedure: (i) transmit the state of the object to the destination in a packed form; (ii) unpack the contents of the received message into the variables of the object; (iii) associate the necessary methods of the object with this object state; and update location bindings to reflect the new location of the object. Our design permits several types of object caching strategies to be experimented with. We exploit the fact that ObjectState servers (which can be Arjuna objects) can be designed to exercise concurrency control over ObjectState instances they are maintaining, and thereby implement any form of cache consistency scheme. In a simple scheme, clients can obtain local copies of an object and access it in a read only mode by first reading locking the ObjectState instance at the ObjectState server. By writing locking, a client can prevent other clients from obtaining local copies (further, the ObjectState server can send messages to clients with old cached copies to invalidate them: for fault-tolerance and consistency, this must be done atomically, using the reliable multicast service mentioned earlier). Such caching schemes would be the principal means of improving performance in our system. Further, by having several ObjectServers for a given class, passive instances of objects can be moved around for load sharing purposes.

In conclusion, we hypothesize that the ability of the underlying operating system to support client-server programming along the lines discussed at beginning of this section provides an adequate support for building persistent object systems. Advance features required for supporting replication and migration (caching) can be built on top, by providing servers, such as ObjectState servers, and by sharing responsibilities between the object management layer and the object system itself. We have indicated how this may be done for the computations controlled by atomic actions. Our prototype implementation provides a basis for testing this hypothesis further as we investigate the performance of our system.

6. Acknowledgements

We are grateful to Graham Parrington and Stuart Wheater for their help with the Arjuna system and to Bart Veer of Perihelion Software for his help with the Helios system. This work has been supported in part by an SERC/IED grant no. GR/F 38402, Fault-tolerant Multiprocessor Systems.

References