SHADOWS - A Flexible Support System for Objects in Distributed Systems

S. J. Caughey, G. D. Parrington and S. K. Shrivastava

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

S.J.Caughey@newcastle.ac.uk
Tel: +44 91 222 7873
Fax: +44 91 222 8232

FULL PAPER - work in progress

Abstract

Shadows is a simple, but flexible, architecture based upon only three key concepts: object servers, object migration and location-transparent operation invocation. We show how several powerful object properties such as object caching, object shareability, and persistence can easily be created by exploiting only these three concepts. The Shadows architecture requires only capabilities found in common object-oriented languages and modern operating systems. An instance of the Shadows architecture has been implemented in C++ on a distributed memory multiprocessor system.

Keywords: Object Migration, Remote Invocation, Persistence, Distribution, Fault Tolerance

1. Introduction

Many systems have been designed to provide support for object-based programs in a distributed system. However, most do so at some sacrifice of flexibility for the object programmer. For example, by imposing the restriction that only persistent objects may be shared, or requiring some combination of specialised hardware, operating system or language. In contrast, this paper introduces the Shadows system that is flexible and based upon the concept of providing fundamental support for only three key facilities - object servers, object migration and the location-transparent invocation of operations upon objects. Using only these three basic facilities the Shadows system permits the creation of other more powerful object capabilities. For example, our prototype C++ implementation (consisting of a number of C++ classes each providing a particular useful property: shareability, persistence, concurrency-control, etc.) allows C++ programmers to create objects that can satisfy various properties both individually and collectively through

a simple use of multiple inheritance. Thus objects that can be shared need not be persistent or even concurrency controlled (although they could be). Furthermore, these properties can be selectively enabled and disabled at run-time based on application requirements. Equally important, the implementation requires no specialised hardware, and uses only standard UNIX-type operating system facilities and standard C++.

The design and prototype implementation of Shadows has come about as a result of our efforts to adapt an object-oriented fault-tolerant programming system - *Arjuna* - developed by us primarily for workstation-based distributed systems [Shrivastava91, McCue92] to run on a locally distributed multiprocessor system. Our approach was to implement a general purpose object management layer (described in [Caughey92]) which exploits the specific features of the underlying operating system - in our case this is the Helios distributed operating system [Perihelion91] running on multiple transputers - and then to use this layer to support the functionality required by *Arjuna*. Careful examination and redesign of this object management layer has led to Shadows, which we believe provides a better way of supporting objects. Although our current implementation functions on a distributed memory multiprocessor system, implementations on alternative platforms, such as networked workstations are clearly possible. This comes about since Shadows only requires a few basic capabilities from the underlying operating system. In particular, we only require a means of creating processes on particular nodes in the system and some form of interprocess communications channel, both of which are either available or can be easily implemented in modern operating systems.

The next section describes the Shadows architecture itself followed by some simple examples illustrating the flexibility that the architecture provides. Implementation details and some initial performance figures are given and finally a comparison is made with other object support systems and some conclusions drawn.

2. The Shadows Architecture

Our architecture is based on the now well known Client/Server model in which remote access to objects is provided by server processes executing on the remote node performing the required operations on behalf of the client. The communication between the client and any servers takes the form of Remote Procedure Calls (RPCs) - an equally well understood extension of a familiar programming paradigm to a distributed environment.

Objects in Shadows are fully encapsulated entities that consist of private internal state and a set of operations by which that state can be accessed and modified. Each object is an instance of a specific class. Shadows is based on the provision of object servers, location-transparent access to objects and on the ability to migrate objects around the system. We argue that these three simple facilities provide adequate basis for supporting a number of other important object properties, including *persistence* (the object outlives its creating process), *shareability* (the object may be accessible to more than one client), and *concurrency*

control (to control access by multiple clients). These, and other user-specified, properties can be used selectively by a programmer to meet application requirements.

2.1. Object Servers

An object server on a node provides clients with access to a set of objects located at that node. In general, servers could provide access to single or multiple objects and, if the latter, to single or multiple classes of object. Using a server per object is heavyweight and makes sharing of objects difficult since the servers will typically have disjoint address spaces. In contrast, allowing a single server to manage objects of more than one class usually requires the compiled operations of each class to be linked into the server and often results in large executables with interdependencies which make later upgrades difficult. Thus, in our prototype implementation, we have adopted an interim solution where each server manages multiple objects of a single class. However, since providing a single server for all objects of a particular class across the whole system would not be scaleable any number of such servers may be instantiated where required.

Servers can be generated automatically by a pre-processor (similar to that described in [Parrington90] which reads the original client class description and generates a corresponding server class; instances of which are responsible for managing objects of the original client class. So, for example, objects of class `Spreadsheet` will be managed by objects (i.e. servers) which are instances of the server class `SpreadsheetServer`. Servers are *active* objects, providing a thread of control that listens for operation invocation requests. Each object server has a unique system-wide *server identity*: knowledge of this identity is sufficient to allow client access to the server. Furthermore, each object managed by a server has a unique name within that server by which it is internally known. The combination of server identity and object name are known as the *system identity* of the object and this suffices to uniquely identify an object within the system. Knowledge of the system identity of an object is considered to be sufficient to allow access to the object.

Object servers have an independent existence and can be instantiated anywhere in the network (initial placement being under programmer and/or system guidance). Servers can also be automatically created whenever the system determines the need. Thus if a server for a particular class is not in existence when a client request is made Shadows automatically instantiates one. This process is not visible to the client (except that the invocation will take longer to complete). Servers terminate whenever they are no longer required, typically when they are not currently responsible for any objects and have not been accessed for some specified period of time.

2.2. *Location-Transparent Invocation*

The second key feature of the Shadows architecture is location transparent operation invocation. Using this mechanism operation invocations may be either handled locally, or, if the object is remote, will be sent via RPC to a server managing that object. This process is totally transparent to the user.

If the object is managed by a server then at the client the object is represented by a special version of the object, known as the *shadow*. This shadow object acts as a stub or proxy [Shapiro86] forwarding all operation invocations to the appropriate server. If, as in Figure 1, a client, in this case `Client1`, invokes some operation on the *shadow*, an RPC containing details of the invocation is sent to the server that holds the real object (the system identity of the real object is held by the shadow). The server invokes the operation on the real object and the results are returned by the server and the shadow to the client.

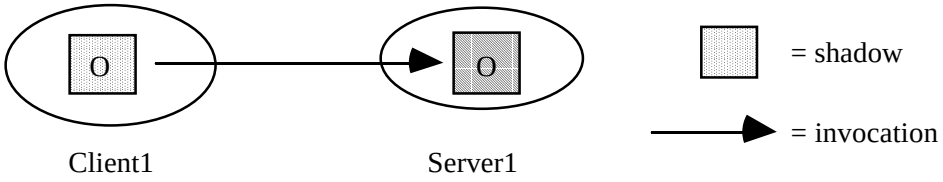

Figure 1: Remote Invocation

Recall that objects may be accessed given only their system identity. This identity may be obtained through a number of mechanisms, such as by use of a naming service like that described in section 3.2.

2.3. *Object Migration*

In a distributed system object movement will be vital to ensure functionality such as load balancing and object placement for improved performance so we have made the provision for object migration a basic requirement of our system. Therefore object servers have the functionality required to both receive and send objects from/to other processes (which may be clients or other servers).

Object migration is based upon the ability to compact an object's internal representation into an architecture neutral snapshot capable of being transferred over the network. Additionally, object servers have support functions that accept new object states for instantiation in the server, or export object states to other servers or clients.

Acceptance of a new state is the job of the `receive` operation. It accepts the name by which an object is to be known and a snapshot of the object's state. Henceforth the object may be accessed at the server by that name. Conversely the `send` operation accepts the name by which an object is known to the server and returns the snapshot of the object state before removing all internal references to the object. Note

that in our architecture only object state is migrated since we assume that the receiving process already contains the required operation code. Furthermore, an object can only be migrated while it is not actively being used.

When an object is migrated from a client to a server, the original object in the client becomes a shadow as outlined earlier. However, the original state still exists in the client shadow and can be used as a cache if required. Server to client migration is the reverse of client to server migration. In this case a client has a shadow of an object (possibly retained following client to server migration) and wishes to migrate the corresponding real object into its address space. To do so the client invokes the `send` function on the server. This causes the object to migrate to the client where it replaces the shadow.

Migration between servers is achieved by simply moving an object from one server to another with no shadows being maintained.

2.4. Object Caching

An important distinction between shadows and stubs or proxies as used in other systems is that in our system a shadow is an instance of the same class as the real object and that shadows may be used (transparently) as caches for the real objects. This means that there are two mechanisms by which remote invocation may be achieved. The first, function shipping, uses RPCs between the client and the server to perform the operation remotely (as described above). The second, data shipping, uses the migration mechanisms to cache a copy of the remote object within the shadow. In this latter case suitable cache coherency mechanisms (for example, those outlined in section 4.2) can be implemented. The choice as to which method to use will typically depend upon expected access patterns.

Note that with only two exceptions, all operations on an object are directed to the cached copy if it is valid, or to the real object otherwise. The exceptions are client migration requests (a client migrating an object to or from a server) and actual caching requests. Both of these requests require co-operation between the shadow and the real object and are always intercepted by the shadow.

3. Exploiting the Shadows Architecture

In this section we will examine how several other useful properties such as shareability, location, and persistence may be provided using the basic object mechanisms of object servers, location-transparent invocation and migration.

3.1. Shareable Objects

All objects in the Shadows system are created in the address space of a particular client and are private to the creating client. However, (providing that they were derived from the `Shareable` class - see the example in section 6) they can be made shareable simply through the process of migration described above. Once migrated to an appropriate server any client that knows the system identity of the object may share access to it. In general this unconstrained sharing often provides no inherent benefit, however, additional benefits can be obtained by utilising some form of concurrency control or migrating the object to a server that provides additional facilities such as making the object persistent as will be shown in a later section.

3.2. Locating Objects

Within Shadows it is sufficient to know only the system identity of an object to access it. However, the system identity, although location-independent, is still server dependent and so will change when an object migrates from one server to another. To accommodate this whenever an object is migrated a *location* object (a simple shareable object) is left behind which points to the new home of the migrated object. Whenever a client attempts to access the object using the old system identity the access will initially fail but the system will automatically re-locate the object by following the references found in the location objects until the object is finally reached whereupon the new system identity is returned and the access completed. Subsequent accesses of the object then go directly to the new home of the object. This location process is completely transparent to the client.

Objects are named using the same mechanism by explicitly creating a location object that points to the home of the object. This location object is then sent to any naming server that manages collections of such objects. The named object may subsequently be accessed by attempting to invoke the required operation on the location object. The invocation is then treated as if the object had migrated from the naming service as described above.

3.3. Persistent Objects

Persistent objects are typically described as having two properties. Firstly, the ability to outlive their creating process which we term *independence*, and secondly the ability to survive system failures with some known (typically high) probability which we term *durability*. On examining these properties we recognised that there were situations in which only one of these properties might be required without incurring the overhead of the other. In particular we envisaged the situation where a set of dynamic, closely co-operating processes are working on some compute-intensive problem and sharing objects. These objects may well need to be shareable but the designer may wish to avoid the overheads of secondary storage access

required by the object being durable. Therefore, we have chosen to implement the persistence properties as two orthogonal properties. The property of independence is satisfied by the property of shareability described earlier. That is, an object sent to, and managed by, a server has the ability to outlive its creating process and does so as long as it remains referenced.

3.3.1. Durable Objects

Durable objects have their states saved on non-volatile (stable) storage. This state must be explicitly saved by executing specific checkpointing operations rather than implicitly saved following every operation on the object, since the majority of systems operate in this fashion. Besides the user may implement the implicit method by carrying out the checkpointing operation at the end of every access if required.

Since the maintained state of an object must be held on some form of (appropriately) stable store, Shadows provides *store servers* responsible for this task. In our current implementation store servers simply hold a copy of the maintained state on disk but other implementations might involve a number of copies mirrored on disk or the use of other secondary storage devices in order to provide the required durability. The maintained state of an object is represented by a `Store` object that essentially holds a snapshot of the state of the object. On creation `Store` objects are migrated to a store server and the state of the durable object given to the `Store` object as the initial durable state. Henceforth any accesses involving the maintained state, such as committing changes or restoring the maintained state, are handled by the `Store` object. This process is illustrated in Figure 2 below.

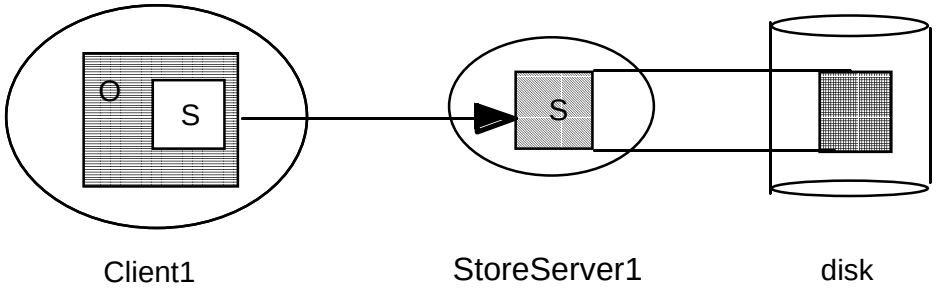

Figure 2: Making an object durable

A client, `Client1`, has an object `O` that is to be made durable; so a `Store` object, `S`, is created within `O`, and then migrated to a store server, `StoreServer1.` The store server will maintain a non volatile copy of the object state, as illustrated. Now any durable state operations are directed by way of the shadow in the client, to the `Store` object which carries out the required store accesses. For example if the client wishes to commit changes made to `O` then the state of `O` is checkpointed and passed to the store object which causes the copy on disk to be updated.

Durable objects survive server crashes. If a server crashes and a client attempts to access an object that was managed by that server then an identical server with the same server identity is started elsewhere in the system. The original access is then re-tried. However, since the new server has no knowledge of the target object the new server optimistically assumes that the object was durable and attempts to obtain the state from its store server. If the object was durable then the store server obliges but it will return the *last saved* state. If the most up-to-date state is required the client must save the state on each operation. The client remains unaware of the server failure (except for the increased access time). If the object was not durable then the store server reports the failure to the server which passes it on to the client. If a store server itself crashes then on the next access to the store server a new store server will be started to respond to the request and will look directly at its secondary storage devices to ascertain if the object was durable.

Shadows thus provides a minimum support mechanism for implementing fault tolerance. More sophisticated mechanisms can be implemented at the application level as demonstrated by systems such as *Arjuna*, that already provides atomicity, concurrency-control, and durability

4. The Flexibility Provided By Shadows

4.1. Mixing Object Properties

Throughout our design we have attempted to make each object property independent of all other properties. This has been achieved by providing suitably generic, low-level facilities; by encapsulating all aspects of an object property entirely within the object and by rigorously maintaining object property independence in the design. This has resulted in a system within which the properties described above, along with other specific properties such as concurrency control, and recoverability, can be used in any combination required by applications.

Our prototype system consists of a set of C++ classes. These classes are organised in a multiple inheritance hierarchy using standard C++ features to allow classes to inherit features provided by other classes. A simplified version of the hierarchy is as follows:

```
Containable
      Shareable
            Store
      Durable
      Recoverable
      Lockable
```

The design of this class hierarchy has been influenced by our earlier work [McCue92]. In general the user will be aware of only a subset of the total number of classes i.e. those offering the user extra

functionality e.g. shareability, durability, recoverability etc. and from these the user may choose a subset at compile-time (through the use of multiple inheritance) for any particular user-defined object class. However, although not always required, all classes in the hierarchy are available to the user for adaptation or replacement as desired. Since each class has been designed to be independent of all others all combinations of properties are feasible. It is our contention that the properties an object requires will often change over its lifetime. Ideally properties could be dynamically bound to objects at run-time but in the absence of such a mechanism the ability to enable and disable properties at run-time has been provided.

In the Shadows system properties inherited by a user-defined object remain inactive until explicitly enabled during run-time. For example, a `Store` object is not sent to a `Store` server (the technique for making an object durable) until the operation `make_durable` is invoked on the object. Subsequently the operations `save_durable_state` and `restore_durable_state` are enabled and may be invoked. At some later stage the property may be de-activated. For example invoking the `make_nondurable` operation on a durable object retrieves the `Store` object from its `Store` server and denies subsequent access to the operations `save_durable_state` and `restore_durable_state`. By this means the inevitable performance overheads and extra resources required by adding functionality are avoided unless the functionality is explicitly requested both at compile-time (through the use of multiple inheritance) and at run-time (by enabling a property).

4.2. Object Caching

A number of different caching strategies are possible within the Shadows architecture. The default method is a simple caching scheme based upon utilising both the `Shareable` and `Lockable` classes. In this case, if a client has a shadow for some object `O` that is currently being managed by a server, then the client may set a read lock on `O` and then cache a copy of `O` locally. Subsequent operation invocations now access the cached copy in the shadow. Other clients may also set read locks and cache their own copies of `O`.

A client obtaining a write lock on `O` before caching it locally has exclusive use of `O` and may write the cached copy back at any time. Note that locking and caching are totally independent of each other and both occur independently on user request.

4.3. Class Replacement by User-defined Classes

The independence of each of the object properties provided means that users may replace the classes which provide those properties with their own versions without having to worry about the impact of the change on other properties. For example the `Lockable` class may be replaced by a class which implements a different form of concurrency-control, the recoverable class may be replaced to implement state-based or log-based recovery, the `Store` class may be replaced to take advantage of different storage media etc.

Similarly, the simple object caching scheme outlined earlier can also be replaced by one that provides more control over clients and is less liable to misuse. In this scheme the caching mechanism is more closely associated with the locking scheme. Using this approach, objects would be automatically cached following the successful setting of a lock, with the cached state being returned as part of the lock request reply message. Indeed caching would be restricted to be valid only under those circumstances. Equally, when the lock was released, the cache would be invalidated automatically, and, if necessary, written back. Many other variations are equally possible.

5. Prototype Performance

A prototype of Shadows has been implemented on a network of transputers running the Helios operating system. The figures below represent some example performance figures from this prototype. These figures were obtained using the spreadsheet example briefly outlined below in section 6. Objects had a size of approximately 100 bytes (ignoring the state associated with distribution support). As a guide to the performance of the underlying system, a simple uni-directional message of 128 bytes using the Helios communication primitives takes 195µs. These were achieved without any optimisation and it should be possible to make substantial improvements on them. In particular the times to commit and restore durable objects are much slower than can be expected as the disk we were using was not connected to our transputer network but to a Sun workstation and had to be accessed by way of the ethernet and a special server on the Sun workstation.

Operation	Elapsed Time
Creating new object	1.7 ms
Local Invocation	3 µs
Migrating object to new server (includes server start up time)	4.5 secs
Migrating object to existing server	135 ms
1st Remote Invocation (includes naming service lookup)	125 ms
Subsequent remote invocation	1.8 ms
Time to cache locally	8 ms
Time to write back cache	8 ms
Migrating object from server	4.6 ms
Time to commit durable object	200 ms
Time to restore durable object	100 ms

6. A Recoverable, Persistent Object with Concurrency Control

The example given below illustrates some of the features provided by our system by describing how shareable, durable, recoverable and lockable properties can be inherited and used by a programmer. Consider the simple spreadsheet class as below:

```
class Spreadsheet
{
    int s[4][4];
public:
    Spreadsheet ();
    void set (int, int, int);
    int  get (int, int);
}
```

The programmer wishes to give objects of this class the properties of shareability, durability, recoverability, and lockability. To do so the programmer must inherit from each of these classes by changing the first line of the code to read

```
class Spreadsheet :    public Shareable, public Durable,
                       public Recoverable, public Lockable
```

The recoverable and lockable classes provide state-based recovery and multiple reader/single writer locking respectively. The code required for the packing of objects and for RPC must now be inserted and `SpreadsheetServer` and `StoreServer` object servers must be generated. Currently the additional code required is inserted manually but the process is relatively simple and could be done automatically by a pre-processor similar to that used by the *Arjuna* system. The property specific functions from each class are now available to the programmer.

At run time a client, `client1`, creates a `Spreadsheet` object, `spr1` - at this stage `spr1` is private to the client, shareability, durability, recoverability and lockability are all currently disabled. All accesses on `spr1` are entirely local and any attempts to access durable or recoverable state, or to lock the object, will return failure indications. In this state the object uses no more resources than a spreadsheet that does not have the selected properties (except for the additional memory associated with the methods providing the properties that were bound into the client at compile time) and the only performance overhead is a single test on a flag (which indicates whether the object is local or remote) on each invocation.

The client now enables recoverability and lockability by invoking the operations `make_recoverable` (inherited from the `Recoverable` class) and `make_lockable` (from the `Lockable` class). Next the client decides to make the object shareable and does so by invoking `make_shareable` specifying the server to be `spreadsheetServer1` and the name of the object to be `spr1`. The spreadsheet is then migrated to `spreadsheetServer1`. Finally `client1` decides to enable durability and does so by invoking `make_durable`. Since the spreadsheet is now held remotely (the version of `spr1` held by the client is a shadow after the migration) the invocation is passed via the shadow

to the real object in `spreadsheetServer1`. The invocation causes the creation of a `Store` object, `S`, which is sent to `storeServer1`, and a copy of the durable state is mirrored on disk. (Note that if any of the above mentioned servers had not already been running the system would have started them automatically.)

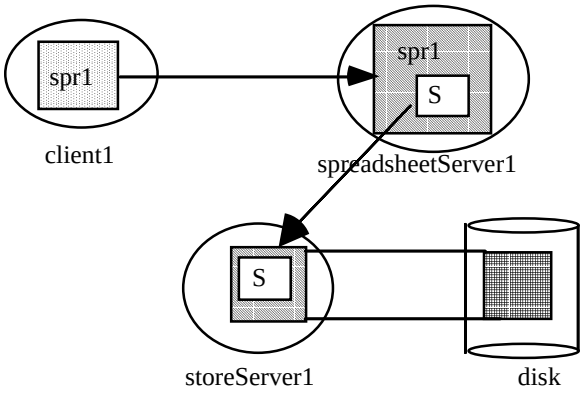

Figure 4: A persistent object with concurrency control and recovery

The state is now as shown in Figure 4. The object is now shared and has recoverability, lockability, and durability all enabled, and the functions relating to all of these properties are now available for use.

At this point the user invokes a `save_recovery_state` operation on the spreadsheet. The object is remote so the invocation is passed via the shadow and `spreadsheetServer1` to the real object. There the state is saved and held within `spr1`.

Now the client invokes `make_nonlockable`, `make_nondurable` and `make_nonshareable` and as a result the system returns to its original state (when `spr1` was first created), except that this time recoverability is enabled and a recovery state is being held. At some later time the client decides to restore the state to that held in the recovery state and so invokes `restore_recovery_state` which restores the object state to that held in the recovery state.

7. Comparisons With Other Systems

Many other object systems that have been described in the literature provide facilities similar to those provided by Shadows (e.g. [Habert90, Liskov83, Shapiro89]). Generally they are not flexible in that they combine one or more of the properties in a fixed manner, whereas we have provided them individually, permitting application specific selection. Furthermore, many are based on new language and/or operating system solutions whereas Shadows is implemented using standard language and operating system. Thus, although the current implementation runs on a distributed memory multiprocessor system, Shadows can easily be implemented to run on say, networked UNIX systems.

The Argus system [Liskov83], for example, effectively provides object servers (Guardians in Argus terminology). However, Argus guardians are far more heavyweight in nature, implicitly encapsulating

recovery and concurrency attributes (amongst others). We could implement Argus style guardians using Shadows by utilising concepts derived from *Arjuna* built on top of the basic Shadows facilities.

The Panda system [Assenmacher93] shares our aims of providing support for distributed, persistent C++ programming through the provision of a set of C++ classes and the use of multiple inheritance to allow the user to choose the appropriate functionality. It also shares in our attempt to move most of the support for distribution and persistence into user space thereby avoiding costly system calls and leaving the system open to application specific enhancement. However, their approach does rely on a specially designed kernel and the provision of distributed shared memory based on 64-bit addressing. In their system memory is statically partitioned over the nodes with hardware support for page faults when a remote address is used. Distributed objects exist in this shared memory with automatic loading of the relevant page into local memory on access. PANDA does have the advantage that it allows direct access to an object's public attributes which our system precludes. Furthermore, they provide thread mobility as an alternate access mechanism. Finally, persistence is provided through the use of a dedicated, shared memory partition mirrored on a stable storage device.

Developed independently, the Inter-Object Communication (IOC) facility [Deshpande92] provides very similar object server and location transparent communication mechanisms to ours with their *secondary objects* and *Liaisons* being equivalent to our shadows and object servers respectively. They too stress the caching capabilities of such a system. However IOC whilst sharing the simplicity of the Shadows architecture is not as flexible in that multiple inheritance is not exploited to the same extent and the system is not as modular, e.g. the naming system is an integral part of the IOC whereas in Shadows the naming system is orthogonal to the communication mechanism. Also, IOC does not exploit the architecture to the same degree as Shadows, for example no support for migration, persistence, concurrency control or recovery has been provided.

A system similar system to Shadows, at least from the viewpoint of its underlying concepts, is the Apertos/Muse system [Tenma92]. This system is also based upon the fundamental concept of object mobility - although Apertos moves objects between metaspaces and not between clients and servers. However, the Apertos designers also suggest that object storage can be implemented by moving an object to a metaspace supporting storage. This is directly comparable with moving the object to a `Store` server in Shadows.

Similarly, the architecture proposed by ANSA [ANSA89] provides the possibility for the selective provision of various object properties including persistence, concurrency control and recoverability. However, these properties are defined at compilation time and cannot be enabled/disabled at run-time. This latter observation is actually true of many proposed systems. In this sense, the Shadows system represents an advance.

All the features described have been implemented and the system currently runs on a small transputer network. In the near future we intend to extend the Shadows implementation to include UNIX

workstations. Longer term we are examining caching strategies and the impact of scale and the integration of mobile hosts (such as laptop machines) into the Shadows architecture.

8. Acknowledgements

This work has been supported in part by grants from the UK MOD and the Science and Engineering Research Council (grant no. GR/H81078), and ESPRIT basic research project No. 6360 (BROADCAST).

REFERENCES

[Ansa89] 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).

[Assenmacher93] Assenmacher, H., T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz, "PANDA - Supporting Distributed Programming in C++", Proceedings of ECOOP93, Kaiserslautern, July 1993. (To appear)

[Caughy92] Caughey, S.J. and S.K. Shrivastava, "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.

[Deshpande92] Deshpande, S., P. Delisle, and A.G. Daghi, "A Communication Facility for Distributed Object-Oriented Applications," Proceedings of the USENIX C++ Conference, pp. 263-277, 1992

[Habert90] Habert, S., L. Mosseri and V. Abrossimov, "COOL: kernel support for object-oriented environments", Proc. of ECOOP/OOPSLA 90 Conf., SIGPLAN Notices, pp. 269-277, 1990.

[Liskov83] 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.

[McCue91] McCue, D.L, and S.K. Shrivastava, "Structuring Fault-Tolerant Object Systems for Portability in a Distributed Environment", Technical Report, Computing Laboratory, University of Newcastle upon Tyne, 1991

[McCue92] 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.

[Parrington90] Parrington, G.D. "Reliable Distributed Programming in C++: The Arjuna Approach", Proceedings of the USENIX Second C++ Conference, pp. 37-50, San Francisco, April 1990.

[Perihelion91] Perihelion Software Ltd., "The Helios Parallel Operating System", Prentice Hall, 1991, ISBN 0 13 381237 5.

[Shapiro86] Shapiro, M. "Structure and encapsulation in distributed systems: the proxy principle", Proc. of 6th Intl. Conf. on Distributed Computer Systems, pp. 198-204, Boston, May 1986.

[Shapiro89] 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.

[Shrivastava91] 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.

[Tenma92] Tenma, T., Y. Yokote and M. Tokoro, "Implementing persistent objects in the Apertos operating system", Proc. of 2nd IEEE Intl. Workshop on Object Orientation in Operating Systems, pp. 66-79, Dourdan, France, September 1992.