# Orthogonally Persistent Support for Persistent CORBA Objects

TR 151

**Adrian O'Lenskie, Alan Dearle and David Hulse**
**Department of Computing Science**
**University of Stirling**
**Scotland**
**{aol,al,dave}@cs.stir.ac.uk**

### *Abstract*

The Object Management Group (OMG) have defined the CORBA Persistent Object Service (POS) which is intended to provide support for persistent CORBA objects. In this paper we describe the CORBA POS and demonstrate how it is deficient in a number of areas. Specifically, it lacks a semantically meaningful failure model, it exhibits problems with encapsulation, and encourages the prevalence of proprietary architectures which inhibit vendor independence. The first of these is perhaps the most serious since it results in the inability of a distributed system to recover to any meaningful state following individual node failures. The paper introduces an alternative CORBA Persistent Object Service hosted by a novel persistent operating system called Grasshopper. Persistent CORBA objects hosted by Grasshopper do not suffer from any of the problems outlined above. The paper briefly describes the Grasshopper system and how it may be used to host persistent CORBA objects.

## *1 Introduction*

The thesis of this paper is that the Persistent Object Service (POS) for the Common Object Request Broker Architecture (CORBA) as defined by the OMG in [13] is deficient in a number of areas. In particular, it lacks a semantically meaningful failure model, its *Persistent Object* abstraction introduces security holes through which objects may be misused, and its imprecise definition encourages proprietary architectures which inhibit open computing. The paper introduces an alternative architecture based on a persistent operating system called Grasshopper [1]. All CORBA objects hosted by Grasshopper benefit from its orthogonal persistence [14] and are consequently persistent regardless of type. Furthermore, the objects exported by Grasshopper are always semantically consistent, even across server failures.

The paper is structured as follows. In Section 2 we give an overview of CORBA. In Section 3 we describe the OMG Persistent Object Service (POS). We detail each of the interfaces defined within the POS specification before examining the service from different perspectives within a network. We will demonstrate the problems associated with the POS and, show that they are mainly due to the lack of support it provides for orthogonal persistence. We describe the basic requirements for orthogonal persistence in section 4. In Section 5 we introduce Grasshopper, and show the advantages of deploying it in a CORBA environment in Section 6. Section 7 describes the Grasshopper CORBA system and Section 8 concludes.

## *2 Overview of CORBA*

The CORBA specification defines an open standard that allows networked objects to communicate across heterogeneous environments. The three basic components in this architecture are:

- a common data representation (CDR), unifying the data model across heterogeneous environments,
- an interface description language (IDL) to provide a means of describing CORBA objects, and
- mechanisms to allow objects to communicate and invoke operations on other objects.

Figure 1 shows how these three basic components combine to form an *Object Request Broker* (ORB), a unifying substrate over which all applications execute. The client and server are CORBA objects and the client wishes to make some request of the server. The IDL stubs provide the indirection required to make the remote access transparent (i.e. the client makes a local request to the IDL stub, which converts this to a remote request). The IDL skeleton allows the ORB to upcall into the object implementation with the request. The object adapter allows the object implementation (the server) to access the functionality provided by the ORB (such as method invocation, security, registration and the generation of object references). The IIOP communication layer provides the transport protocol used to transmit a local request to a remote implementation. For a detailed description of the ORB see [10].

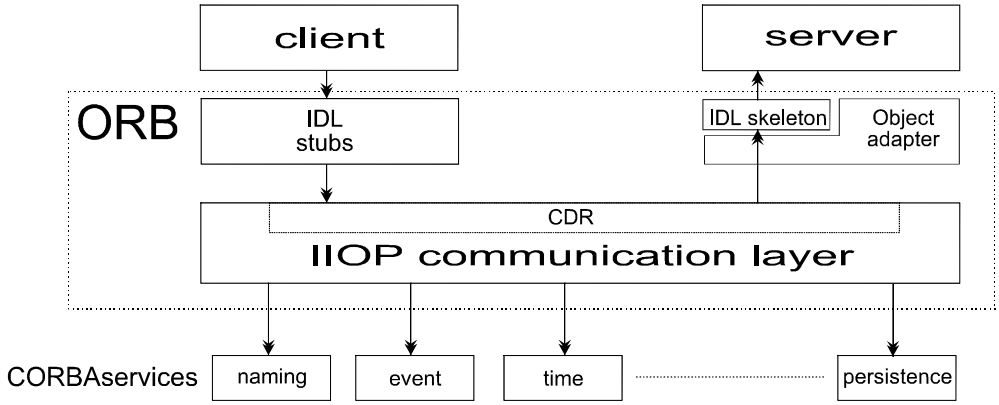


**Figure 1:** The architecture of the ORB

The three components introduced above are all that is required to enable communication between any network components, however additional (optional) functionality that could be useful to a distributed application designer is defined in the Common Object Services (COS) specification (CORBAservices). CORBAservices are connected directly to, but do not form part of the ORB. Figure 1 shows an ORB with *naming*, *event*, *time* and *persistence* services.

CORBAservices are motivated by the fact that most distributed applications have common functionality in addressing problems associated with distribution. Rather than solving the problem separately in each application, the CORBAservices, if present, offer this functionality to all applications using the ORB. The client object can make use of the service interface (defined in IDL) and is unaware of whether the service implementation is local (for example a shared library) or remote. One of the services that are provided is the *persistent object service* (POS) (see [11] for the complete specification of all services).

## *3 The Persistent Object Service (POS)*

The motivation behind the POS is "to provide common interfaces to the mechanisms used for retrieving and managing the persistent state of (CORBA) objects" [OMG, March 31, 1995 #2163]. The specification indicates that the service is intended to be used in conjunction with other services. For example, a distributed database might use the *Object Collection Service* to structure its data, the *Object Transaction Service* to control updates, the *Relationship Service* to maintain dependencies between data and the *Persistent Object Service* to store the data.

The POS is composed of several (independent) abstractions that combine to provide a service:

- Persistent Identifier (PID) – identifies the location of an object's persistent data in a datastore,
- Persistent Object (PO) – an object that supports an interface allowing a client to control the persistence of its state,
- Persistent Object Manager (POM) – redirects the abstract persistence requests from a POS client to a particular mechanism used to control an object's persistence,

- Protocol – provides the interface between the Persistent Data Service (described below) and target object, allowing the state of the object to be extracted and restored,
- Datastore – provides a particular mechanism for maintaining an object's persistent state, and
- Persistent Data Service (PDS) – provides an interface to any protocol/datastore combination. That is, it applies a *protocol* to a *persistent object* in order to store its state in a particular datastore.

The POS specification is exceedingly abstract in places thereby allowing implementations to take almost any form. In other places it is rigid and restrictive. This will become apparent after a description of how the service operates.

#### 3.1.1 Summary of Interaction Between Modules

Figure 2 shows a typical usage of the CORBA POS. In general a client must perform some initialisation before persistence operations can occur. In the example shown in Figure 2, this process starts with the creation of a PID (Figure 2(1)). The PID is constructed through the use of a *factory object* contained within the PID abstraction. The PIDfactory associates the newly created PID with a particular datastore. The PID must then be associated with a particular persistent object. A POfactory object is used to create an instance of a particular PO. The PID is passed to the POfactory along with an indication of which POM the PO is to use (Figure 2(2)).

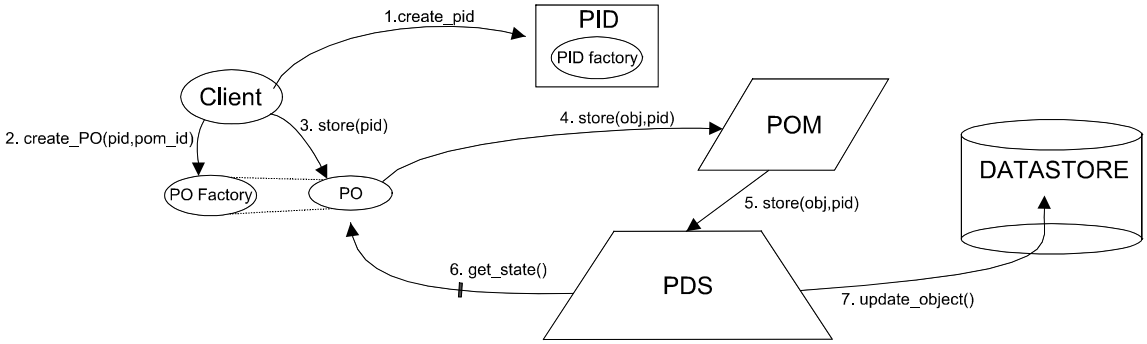


**Figure 2:** The CORBA Persistent Object Service

The client can now perform persistence operations on the PO. For example, if the client wishes to preserve the state of the PO, it issues a *store* request to the object (Figure 2(3)). Since an object may be associated with multiple PIDs (a client may maintain different *versions* of an objects persistent data), a particular PID is passed in the store request. In response, the PO sends a store request to its designated POM (Figure 2(4)). The POM determines which PDS supports the combination of protocol (exported by the PO) and datastore (defined by the client). The POM then forwards the store request to the appropriate PDS (Figure 2(5)). The PDS extracts the state from the PO (Figure 2(6)) using the defined protocol, and stores the state in the datastore using some unspecified interface (Figure 2(7)).

The above example usage is one of many that are possible, however it gives an overview of how the components are expected to interact. Sections 3.1.2 to 3.1.7 provide a detailed description of each module within the POS.

#### 3.1.2 The PID Module

A PID identifies the location of an object's persistent data within a particular datastore (how the POS determines which datastore is not defined). In the simplest case, the POS will support just one datastore for all objects. More complex implementations may provide interfaces to multiple datastores. If there is more than one type of datastore, the client can define which is to be used when requesting a PID (using the *datastore_type* attribute within the PID module). The interface to the PID module consists of two elements: one indicating the type of the datastore used to support the persistent object and the other being a method returning a string identifying the PID.

```
module CosPersistencePID{
        interface PID {
                attribute string datastore_type;
                string get_PIDString();
        };
};
```

### 3.1.3 The Persistent Object (PO) Module

The interface supported by the PO describes the operations available to control the persistence of an object.

Unfortunately the POS specification does not include consideration of a compound object model and consequently does not provide explicit support for the persistence of a graph of objects. However it does allow the persistence of an object to be controlled externally by a client provided that all the objects in the object graph support the PO interface. For example, Figure 3 shows a graph of objects which are to be made persistent. At time *t0*, object A performs a store. The state of A is extracted by the PDS and the references to B and C are detected, and those objects sent a *store* signal. At time *t1*, the state of B and C are extracted by the protocol and any objects referenced are sent a store signal (in this case D). Finally, at time *t2*, the state of D is extracted and the process completes. The client of a persistent object is responsible for:

- creating and initialising a PID for the object,
- binding the PID to the object instance, possibly using some object factory, and
- controlling the relationship between the object and datastore.

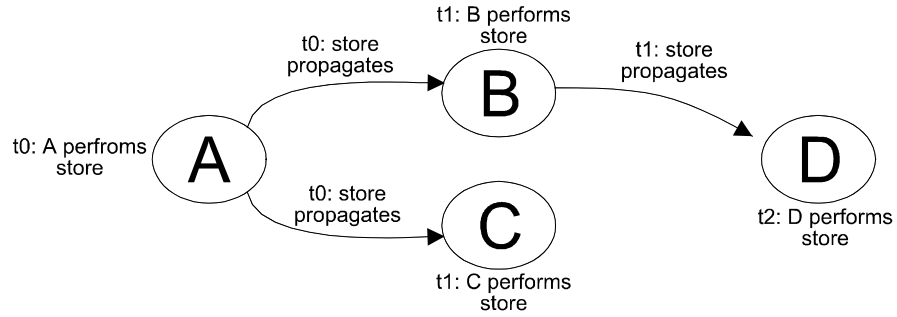


**Figure 3:** Use of the POS to support compound objects

```
module CosPersistencePO {
        interface PO {
                attribute CosPersistencePID::PID p;
                CosPersistencePDS::PDS connect (
                        in CosPersistencePID::PID p);
                void disconnect (in CosPersistencePID::PID p);
                void store (in CosPersistencePID::PID p);
                void restore (in CosPersistencePID::PID p);
                void delete (in CosPersistencePID::PID p);
        };
```

```
interface SD {
                    void pre_store();
                    void post_store();
          };
};
```

The operations supported by the interface are described in the following sections.

*Connect/disconnect*

A client can establish a relationship between a persistent object and datastore with the *connect* call. The *connect* signal propagates through to the POM and PDS which implements the relationship. A *disconnect* call destroys the relationship. The effect of the relationship is to reflect the transient state of the object with its persistent state within the datastore, an operation on the object will automatically update the persistent state. There is no description of how this is to be achieved from the POS specification, however it does suggest that the persistent state within the datastore may be updated after every operation on the object.

*Store/restore*

The *store* and *restore* operations are provided to explicitly move data to and from the store. The actual data that is moved is determined by the implementation of the PO. The *store*/*restore* methods contained in the PO are responsible for determining how much and what data is moved between the PO and the POS.

*Delete*

A *delete* call removes the persistent data associated with a particular object from the store.

*Pre_store/post_restore*

The *pre_store* and *post_restore* operations allow for synchronisation. Tasks that must be undertaken (such as flushing of buffers) before transient data is made persistent are implemented in the *pre_store* operation. The dual of this operation (*post_restore*) implements tasks that must be undertaken after a *restore* operation (such as moving the object to a specific state). The operations are implemented by the object and intended for use by the POM. However, there is an opportunity for encapsulation to be breached as any object can invoke these operations.

#### 3.1.4 The Persistent Object Manager (POM) Module

The POM routes requests from persistent objects to a PDS that supports both the datastore desired by the client and the protocol implemented by the persistent object. A POM may have a fixed number of PDSs and no mechanism for introducing any more, or it may support an interface for allowing other PDSs to be added to the POS (this allows other datastores and object protocols to be supported). Such an interface is not defined by the specification.

```
module CosPersistencePOM{
        interface Object;
        interface POM {
                CosPersistencePDS::PDS connect (
                        in Object obj, in CosPersistencePID::PID p);
                void disconnect(in Object obj, in CosPersistencePID::PID p);
                void store(in Object obj, in CosPersistencePID::PID p);
                void restore(in Object obj, in CosPersistencePID::PID p);
                void delete(in Object obj, in CosPersistencePID::PID p);
        };
};
```

The methods implemented by the POM are called by the PO, and call operations of the same name in the PDS.

#### 3.1.5 The Protocol Module

The interaction between an object and the PDS is termed the *protocol*. It may consist of calls to the object, calls from the object, operations implemented with hidden interfaces, or some combination of these. The POS specification does not define any aspect of the protocol, but does give example protocols, one of which is the *Direct Access* protocol. This uses the underlying CORBA mechanisms for managing an object's attributes to expose these attributes to the POS.

#### 3.1.6 The Datastore Module

The interaction between the PDS and the actual data repository is termed the *datastore*. Again, as with the protocol, no aspect of the datastore interface is defined by the POS specification. The specification does show an example interface describing how it might be achieved.

#### 3.1.7 The Persistent Data Service (PDS) Module

The PDS module maps (a specialisation of) the abstract persistence operations received from a client (via the POM) onto the protocol on the client's side, and a datastore interface on the repository side. This has the effect of de-coupling the datastore from the object, that is the object could implement its protocol using the *externalization service* [12], and this could be used to map the object's persistent data onto many different types of datastore.

```
module CosPersistencePDS {
        interface Object;
        interface PDS {
                PDS connect (in Object obj, in CosPersistencePID::PID p);
                void disconnect (in Object,in CosPersistencePID::PID p);
                void store (in Object obj, in CosPersistencePID::PID p);
                void restore (in Object obj, in CosPersistencePID::PID p);
                void delete (in Object obj, in CosPersistencePID::PID p);
        };
};
```

### 3.2 A Client's View of the POS

Figure 4 shows the view a client (of a persistent object) has of the POS. A client requires a PID before any persistence operations can be performed (Figure 4(1)). The PID is associated with the PO (possibly through the use of a POFactory) and the POM to be used is specified (Figure4(2)). As shown in Figure 4(3), the client can now use the persistence operations as supported by the PO, such as *store(pid)*.

Unfortunately, due to the nature of the POS specification, a client must deal with a number of problems. These are described in sections 3.2.1 and 3.2.2.

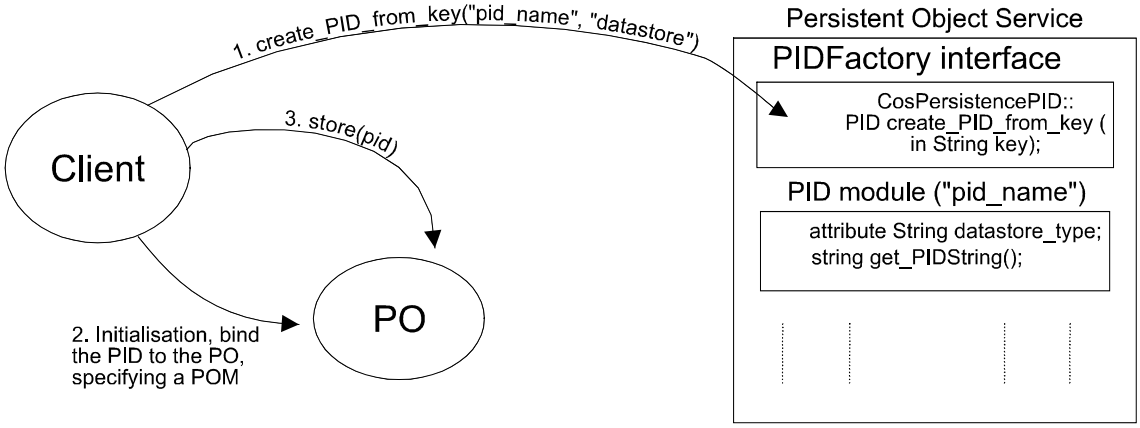


**Figure 4:** The client's view of the Persistent Object Service

#### 3.2.1 No Failure Model

Although the client in this model is the instigator of persistence operations, it is here that its involvement ends. There is no mechanism defined for reporting status or results back to the client. This is essential if the POS is being used for resilience. For example, Figure 5 depicts a client managing the persistence of two objects (A and B) which represent *children*. Both children are in the same school class and as such they share the same *teacher* (represented by Y). Although both objects maintain their state independently, they conceptually share state which is part of their persistent data (here it is a reference to their teacher, Y). The integrity of the (distributed) system depends on each child of a particular school class having the same teacher. In Figure 5(1), the state of the persistent store is consistent with the transient objects. Figure 5(2) depicts the class with a new teacher (Z) and the client updates both objects to reflect this. A store operation is performed on both objects by the client to make the persistent data consistent with the transient state of the objects. A failure at any point of the store operation on either object would leave the persistent data inconsistent with the transient objects. If the failure is limited to the store operation (i.e. other persistence operations result in success), the opportunity for failure is compounded. In Figure 5(3), the client performing a restore operation causes the inconsistency to be back propagated into the transient system resulting in a globally inconsistent state. In our example, it results in two pupils in the same class being taught by different teachers.

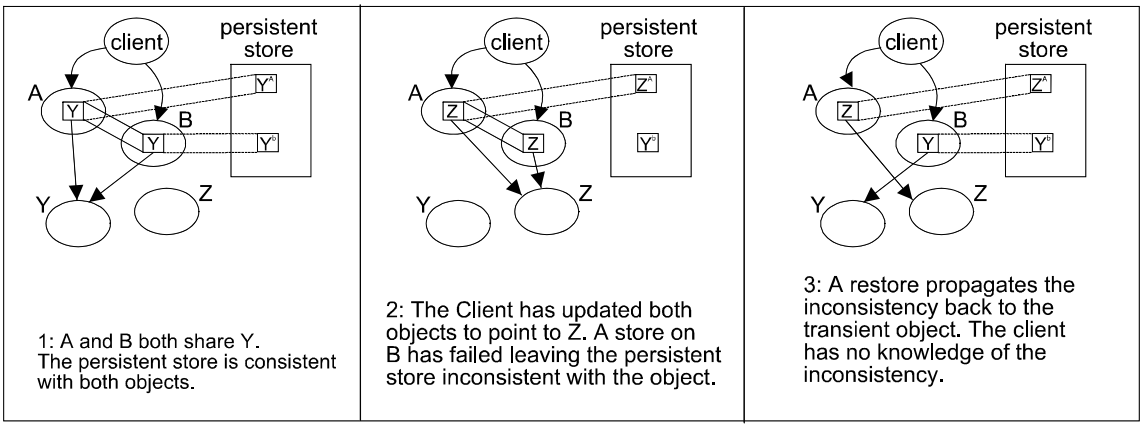


**Figure 5:** Semantic inconsistencies

Such situations are typically avoided through the use of transaction control mechanisms. The client requires the ability to undo all *store* operations if one *store* fails. OMG's Object Transaction Service (OTS) [12] is defined for exactly this purpose. However it cannot be used to control the use of the POS by a client, since the critical resource being manipulated (some entry in a datastore) is disjoint from the source of the request (a client of a PO).

### 3.2.2 Local Persistence Operations Yield Global Inconsistencies

The motivation behind allowing a client to control the persistence of an object is to allow persistence operations to propagate through a distributed system. If an object is to be made persistent, it can broadcast a *store* operation to all the objects on which it depends. For example, Figure 6 shows the chain of command within a company. Each object maintains state to be made persistent (p) and state that may be transient (t). A is a company director, B reports to and gets instructions from A, C reports to and gets instruction from B, and D usually reports to and gets instruction from C. These relationships are captured within the object's persistent data. For a limited time however, D is seconded to work immediately under the director. To model this, A maintains this relationship within state not included in persistence operations (t). Figure 6(1) shows how a *store* operation on A will cause a *store* on B, which in turn causes a *store* on C, and finally a *store* on D. In Figure 6(2), A is pleased with D's work and modifies the state of D to indicate that he is ready for promotion (p1 is updated to p2 and t is updated to t2). Meanwhile, B has detected some failure in C (but cannot decide whether the problem lies with C or itself) and decides to perform a *restore* operation on itself to restore the last known consistent state, shown in Figure 6(3) Just as B's *restore* implementation propagates a *restore* signal to C, so C's implementation propagates the signal to D. All updates made on D since the last *store* operation have now been lost, D is no longer ready for promotion. More crucially, A is unaware of the adoption of earlier state by objects it depends on. A is now inconsistent with the rest of the system.

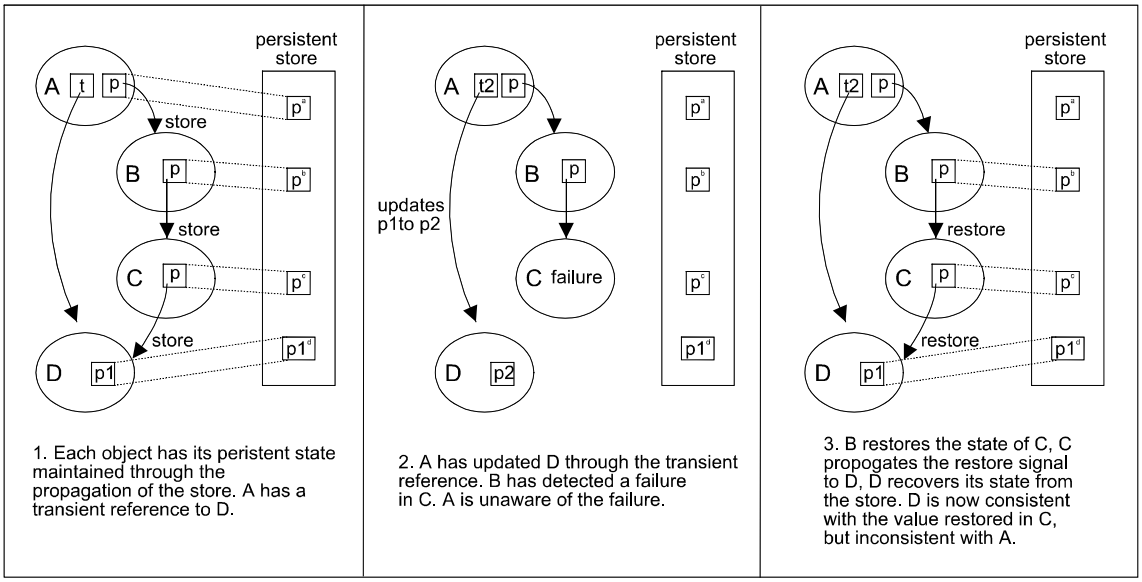


**Figure 6:** Lack of global control

The problem arises because it is not possible to notify the client of a persistent object that a *restore* has been performed on that object. One solution to this problem would be to define a mechanism to allow a *restore* signal to be broadcast to clients of an object. However, there is no mechanism defined for identifying the clients of a persistent object.

## 3.3 An Object's View of the POS

Figure 7 shows an example of how a persistent object views the POS. In this example, a client is managing the target object's persistence, the object must therefore support the PO interface. (An object may decide not to support the PO interface and manage its own persistence operations). Any persistence operation (in this case a *store*) is passed to the POM. The POM resolves which PDS to use (the PDS must support the protocol implemented by the object and an appropriate datastore interface). The *store* request is forwarded to the PDS, which extracts the persistent state from the object using the implemented protocol. The following sections describe a number of problems that arise and are highlighted by the example shown in Figure 7.

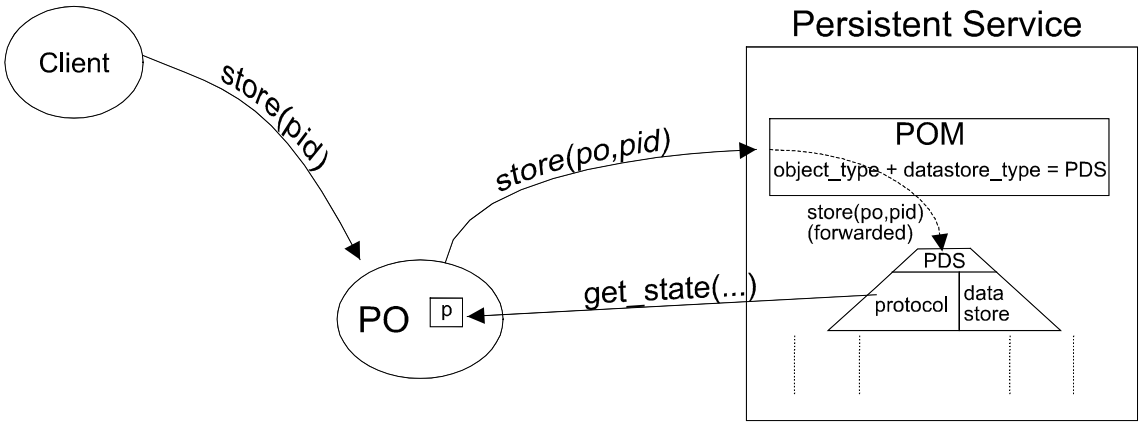


**Figure 7:** Persistent Object's View of the Persistent Object Service

#### 3.3.1 Legacy Objects

The server object (the PO) must inherit from a known protocol if it is to use the POS. This restricts the use of the POS to objects that expect to use it at compilation (definition) time. Therefore the range of objects that an application can use within a persistent system is limited to those explicitly designed to support persistence.

#### 3.3.2 Loose Association Between PID and Object

Another problem for a PO is the loose association between an object and its PID (and therefore its persistent data). For example, in Figure 8 objects A and B share a reference to object C and manage its persistent state independently (possibly even in different datastores). After some failure, both objects attempt to reconstruct C using the persistent data (possibly through the use of some generic object factory). As neither recognises the object is shared, two objects are created. This problem is known as loss of *referential integrity*.

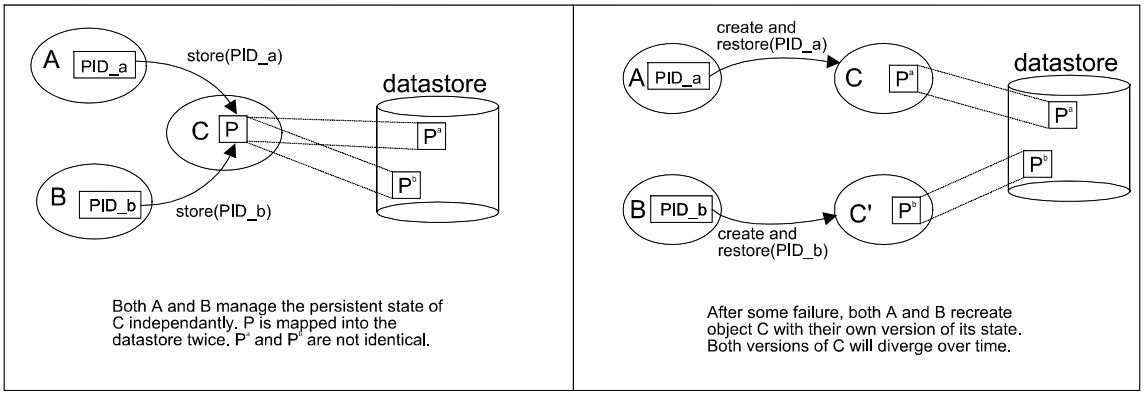


**Figure 8:** Weak binding between PID and object

#### 3.3.3 Broken Encapsulation

A PO cannot arbitrarily decide which protocol to use to export its state to the POS. It must use a protocol implemented by the POS. This can lead to broken encapsulation of an object thereby seriously breaching security. If the POS supports the most basic protocol (the Direct Access protocol discussed in Section 3.1.5) any client of a PO has the opportunity to use this interface to circumvent encapsulation.

#### 3.3.4 Proprietary Architecture

The PDS module that is used to extract the state from an object and enter it into some datastore underpins the Persistent Object Service. The specification does not define any requirements for either of these activities. More importantly, the specification does not indicate how a POS is to support the ability to add new PDSs. Without at least this defined, there is no guidance to POS designers that will allow implementations of different modules to interact in a single POS.

### 3.4 Summary of the POS

The problems introduced above result from a client applying persistence operations to transient objects. It has long been argued that persistence should not be obtained through explicit operations on an object, but should be an intrinsic attribute of an object, supported by the infrastructure within which the object operates [2]. That is, the persistence of an object should be *orthogonal* to *use*, *type* and *identification* of the object.

## *4 Orthogonal Persistence*

The three basic principles behind orthogonal persistence [9] are that:

- all objects may (persist) exist for as long, or as short, a period as is required,
- all objects are manipulated in the same manner, regardless of their longevity, and
- the identification of persistent objects is not linked to the type system.

All objects have the right to the same persistence regardless of type; a system only supports type orthogonality if the range of time that an object can persist is the same for all objects. This means a particular set of data is guaranteed to behave in a consistent manner as each of its elements posses the same persistence. A designer is never faced with the situation where one element of a data set can persist while another cannot.

All programs look the same regardless of the longevity of the data over which they operate. This means the operations to operate on transient data are the same as operations required for long-lived data. This allows the designer to concentrate on the problem domain. For example, the designer of a simple text editor can reason about the functionality of the editor without adding the complication of miscellaneous operations such as performing incremental saves. These operations are not in the domain of the problem the programmer is attempting to solve. The underlying support system must implement the mechanism required to move data to/from store and perform any required translation, all transparently to the programmer.

If the persistence of data is identified through the use of specific variable names, or specific types, then all objects are not treated uniformly, contravening the first principle. The technique used to identify persistent data must be independent of the type system. One such method would be to employ *persistence by reachability*. The system automatically identifies persistent data by following pointers from some persistent root[3].

OMG's persistent model provides limited support for the first principle. Indeed this support is somewhat less than a traditional operating system could offer (a write system call can report failure to the instigator of the call, whereas a *store* operation cannot). Without orthogonal persistence, it is impossible to solve any the problems identified with the Persistent Object Service.

There have been two approaches to providing orthogonal persistence, the first is to provide persistence through the language system [3] and the other is through the operating system [5]. Within a CORBA environment, the language system may vary throughout an application. For example, an application might choose to implement data manipulation operations in C, a graphical front end using Visual Basic, provide Internet access to the application using Java, and other elements of the application using other languages. For the application to enjoy the support of orthogonal persistence at the language level, all of the languages implementing the objects in the application must support persistence. As there is little support within host operating systems for persistence, most persistent language designers implement a *persistent abstract machine* above the operating system. If each language implements its own persistent abstract machine, much effort is duplicated. With regards to the CORBA environment, each of these abstract machines would require its own implementation of the object request broker.

Implementing orthogonal persistence within the operating system avoids these problems. We have constructed the Grasshopper operating system to research the implications of providing persistence as a basic operating system abstraction. Grasshopper executes on conventional hardware platforms and presents the user with a resilient and persistent environment. Applications can be built using standard language technologies (assembly language, C, Napier88, C++ and Java) and can be executed within the Grasshopper environment. In sections 5, 6 and 7 we discuss the abstractions provided by the Grasshopper operating system, and show how Grasshopper may be used in the CORBA environment.

# 5 The *Grasshopper* Operating System

Grasshopper is an example of a persistent operating system. In this section we describe the three basic abstractions provided by Grasshopper. The abstraction over storage is the *container* and the abstraction over execution is the *locus* (plural *loci*). The third basic abstraction is the *capability* which provides control over access to Grasshopper entities.

## 5.1 Containers

Containers are the only storage abstraction provided by Grasshopper; they are persistent entities which replace both address spaces and file systems. Like address spaces and files, containers may hold both code and data and sometimes both. In most operating systems, the notion of a virtual address space is associated with an ephemeral entity, a process, which accesses data within that address space. In contrast, containers and loci are orthogonal concepts. A Grasshopper system consists of a number of containers which may have loci executing within them. At any time, a locus can only address the data visible in the container in which it is executing – its host container. However, an arbitrary region of one container may be mapped into another. This provides a basic mechanism for sharing data between containers and for the construction of libraries. Each container may export an *invocation point* which permits a locus to change its execution environment to the container and begin executing the code found at the invocation point. When used in conjunction with containers containing both code and data, this provides a mechanism with which hardware protected abstract data types may be constructed. In Grasshopper many containers export IDL interfaces which are used by invoking loci to obtain services provided by the container.

## 5.2 Loci

A locus is essentially a kernel-supported thread that can change its execution environment by *invocation*. Thus, a locus may invoke a container thereby changing its host container and may later return to the original container. Like containers, loci are maintained by the Grasshopper kernel and are inherently persistent. A locus is associated with a *host* container. The locus perceives the address space of the host container overlayed with any mapped containers. Virtual addresses generated by the locus map directly onto addresses within the host container and any privately mapped containers. A container comprising program code, mutable data and a locus forms a basic running program. Loci are an orthogonal abstraction to containers; any number of loci may execute simultaneously within a given container.

## 5.3 Protection

In the previous sections we have described two of the three basic abstractions in Grasshopper and the operations over these abstractions. Given that containers are the *only* abstraction over storage (i.e. there is no file system), a naming and access control mechanism is required and this is provided by *capabilities* which are protected by segregation. For a full description of capabilities see [6].

## 5.4 Naming

Symbolic naming which maps from strings to capabilities is provided by *nameservers* which are themselves constructed from user-level containers using the mechanisms described above. Nameservers support attributed mappings from strings to capabilities. An arbitrary number of typed attributes may be associated with each mapping. Like most Grasshopper entities, nameservers are implemented as abstract data types containing the nameserver code and the data they support. They export an IDL interface which permits loci to invoke them in order to make bindings, iterate over their contents, and extract capabilities and attributes.

### 5.5 Recovery

The Grasshopper kernel treats loci and the data accessed by them during computation as the unit of recovery. Loci are able to snapshot the state of their computation at any time, a task which is co-ordinated by the kernel and draws on services provided by user level managers to snapshot user level data. A snapshot consists of all the data related to the computation of a locus and includes:

1. any modified container data seen by the locus,
2. any data maintained within the kernel to represent the state of the locus (including the registers) and the containers in which it has executed.

Since a locus can move between containers during the course of its computation, a snapshot typically involves recording the state of pages within a number of different containers. In contrast to other persistent systems in which a snapshot involves making the entire persistent store stable, the snapshot mechanism in Grasshopper only affects the stability of the portions of containers seen during the computation of a particular locus. Since loci are free to use shared memory as a means of inter-process communication; the actions of one locus can be influenced by the actions of another. This interaction creates causal dependencies between loci. During the normal operation of the system it is possible to ignore these causal dependencies because they are automatically preserved. However, if the system needs to be restarted after a shutdown or crash, locus snapshots must be used to rebuild a consistent system state.

It is therefore necessary to detect causal dependencies and ensure that they are preserved across failure of the system, thus guaranteeing global consistency. Detection of causal dependencies is performed by the kernel and managers which monitor read and write faults to compile modified page lists containing an entry for every modified page seen by a locus since its last snapshot. In addition, the kernel also maintains a list of containers in which a locus has seen modified data. The kernel uses this list to determine which managers it must request to snapshot data modified by the snapshotting locus.

The kernel co-ordinates the processing of locus snapshots and maintains dependency information such that it is possible to recover the state of the system from a causally consistent set of locus snapshots following a failure. Causal dependencies between loci are represented using vector time [7].

The above mechanism guarantees that a Grasshopper system will always recover both data and computation (loci) to a globally consistent state. It does not guarantee that the snapshotted state of the system was semantically consistent when a snapshot was made. Such guarantees require either co-operation or exclusion at the application level. Grasshopper supports these activities by providing mechanisms that allow concurrent loci to co-operate. These mechanisms include semaphores and conditional locks.

## *6 Using Grasshopper in a CORBA Environment*

The CORBA specification includes no formal semantics for the persistence of an object. With no guidance, distributed application developers will adopt ad-hoc descriptions of their behaviour. In particular, the specification does not describe behaviour regarding problems associated with the nature of transient objects. Implementations are at liberty to deal with all such problems in an arbitrary fashion.

The basic premise behind Grasshopper is that every object has the right to persist and be resilient against failure. Within a distributed context, this statement requires further qualification – after all, a platform running Grasshopper suffers just as much during a power failure as any other system. Figure 9 shows a typical sequence of events that might occur with objects based in a traditional “non-persistent” environment when there is a power failure.

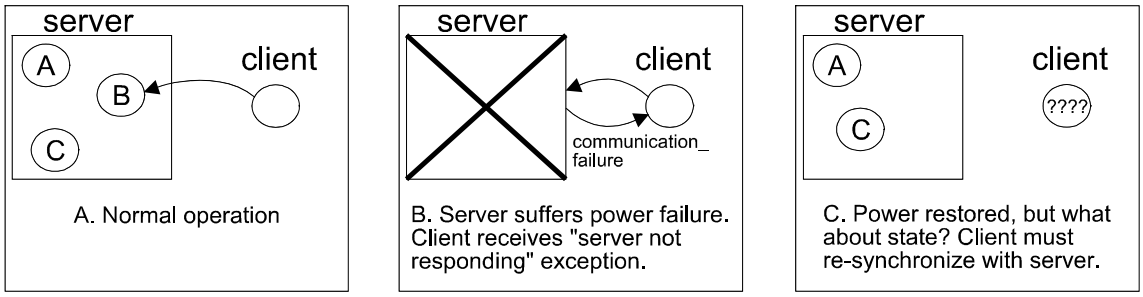


**Figure 9:** Failure with CORBA Persistent Objects

Figure 9(A) shows the client accessing object B on the server. In Figure 9(B), the server is experiencing a power failure and the client is informed of the symptoms of the problem by the underlying CORBA exception model. The client is unsure of the cause of the problem, any component between itself and the server could be at fault. If it were a transient communication problem, the server state would be maintained, although the client is not at liberty to make these assumptions. In Figure 9(C) power is restored. However some of the state has been lost. The state that survives is that which has explicitly been made persistent and the server automatically regenerates. The behaviour of the client is tempered by the failure; that is the client is explicitly aware of some problem and must participate with the server in recovery. Thus the client must support *recover-on-failure* semantics. This may involve re-synchronising its state with that of the server object. Compare this to the same scenario within a Grasshopper environment (shown in Figure 10).

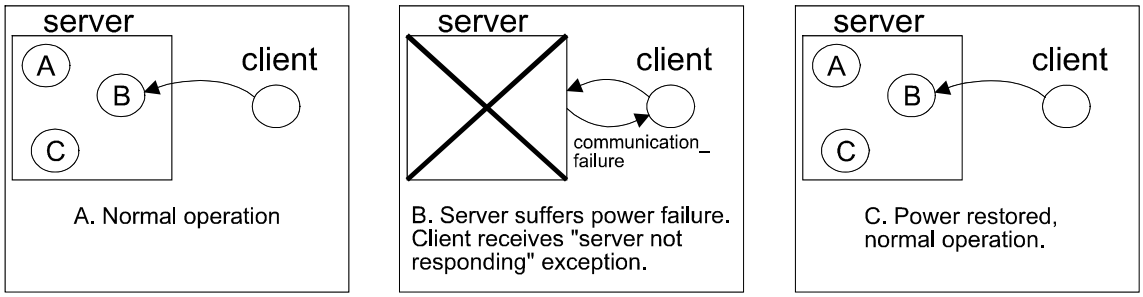


**Figure 10:** Failure with Grasshopper Persistent CORBA Objects

The behaviour the client experiences is identical to that of the traditional non-persistent system up until Figure 10(C). As Grasshopper guarantees the restoration of the state of all objects operating within its environment to some consistent state, the client may safely treat all failures as transient communication problems. When the power is restored, the state of the server will be restored. This results in a system that is easier for the client to reason about. The client only has to support *delay-on-failure* semantics. This is the ideal situation, however due to certain Byzantine conditions, there is likely to be some minimal re-synchronisation required.

## *7* Provision of (Persistent) CORBA Objects Using *Grasshopper*

Figure 11 shows the implementation of the CORBA system on Grasshopper. The functionality the *arbitrator* provides is similar to that provided by the *Orbix Daemon* (orbixd). The arbitrator acts as the broker permitting clients to connect to objects and servers hosted by Grasshopper. In order to do this it must be able to discover which containers want to export CORBA services. The arbitrator manages a namespace with which loci can register containers offering services (Figure 11(1)). This namespace is consulted whenever the arbitrator receives an incoming request.

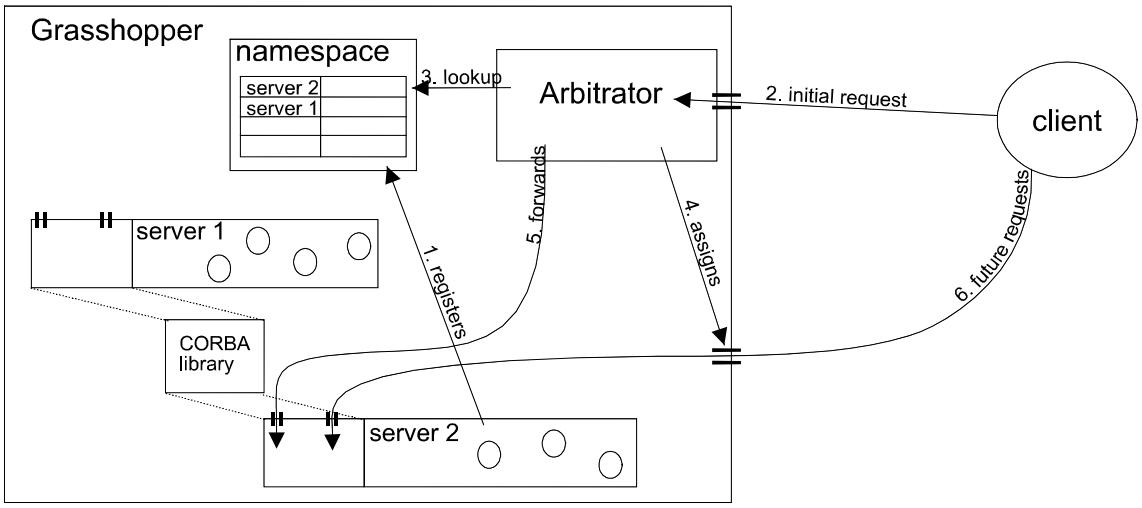


**Figure 11:** The Grasshopper CORBA System

The act of servers registering their existence with the namespace is similar to the *putit* utility provided with Orbix. The granularity with which an object can register with the arbitrator is currently on a per-server basis (equivalent to the *shared activation mode* of Orbix). In the future we expect to extend this to include per-object and per-method registration. The identity of objects hosted by Grasshopper is supported by a two-part naming scheme. The first half identifies a container within the nameserver, the second a local name within the container. Local names within containers are implemented as indexes into an *export table* implemented within the container. This permits access controls to be implemented locally by the container. It also prevents all objects from being inadvertently visible to the world.

Every container registered with the nameserver provides a standard set of entry points with which the server can invoke the container. This provides the basic mechanism for activation of a CORBA server. The containers contain generic CORBA code, application specific code and objects. The code implementing the CORBA code is provided by the Grasshopper CORBA library which is mapped into server containers. This library (which is also used by the arbitrator) contains the IIOP engine, the dispatcher and assorted necessary functions.

An incoming request (Figure 11(2)) arriving at the arbitrator is handled as follows:

- the arbitrator resolves to which server the client is attempting to connect (Figure 11(3)),
- a locus is either taken from a pool or is created to handle the incoming request (the arbitrator is the source of computational elements),
- the server is assigned an as-yet unused communication port which is used for further communication with the client (Figure 11(4)),
- the server is invoked with the IIOP packet and communication port as parameters (Figure 11(5)), and
- future requests occur on the newly assigned port (Figure 11(6)).

Once the container has been invoked, the code at the invocation point calls code contained in the CORBA library to decode the IIOP packet. The target object is looked up in the container's export table and provided that the object is found, the dispatcher will call the correct object with the parameters extracted from the IIOP packet. The CORBA library is responsible for maintaining the appropriate state to enable the reply to be returned to the client on the port passed to the container by the arbitrator.

## 7.1 Persistence of CORBA Objects in Grasshopper

In the above discussion, persistence, the main topic of this paper has not been mentioned. This is because in an orthogonally persistent system, persistence is an ever present but invisible property of the objects and, in the case of Grasshopper, the processes. As described above, Grasshopper supports a strong causal

consistency model that ensures that all computations performed on Grasshopper are always causally consistent. However, if the problems of consistency described sections 3.2 and 3.3 are to be addressed, some mechanism needs to be provided to ensure that inconsistent states are not visible from outside the Grasshopper system, in this case by CORBA clients. In Grasshopper, this may be addressed in a number of ways which basically trade cost against implementation efficiency.

The simplest approach to ensuring view consistency is to snapshot the state of loci manipulating CORBA objects after every update and before the update has propagated to CORBA clients. This approach may be sophisticated enough for many applications since snapshot costs are low, especially when performed frequently due to Grasshopper's log structured storage technology employed by the kernel [8]. However, it is also possible to log incoming requests and use these to restore the system to an externally visible state during recovery. This approach depends on the ability to restore a computation to some snapshotted state. The recovery manager can pull the state of the container forward in time to the state perceived by the clients by replaying the messages from the message log. The recovery manager must ensure that any outgoing messages are discarded during recovery to maintain the consistent view held by clients.

## *8 Conclusions*

OMG's Persistent Object Service specification is intended to allow distributed application developers to use a standard interface to control an object's persistence. Reasoning about the persistence of an object (especially in a distributed context) is difficult, and the specification has done little to aid this. We have shown that the POS lacks some core functionality resulting in the following problems:

- no failure model resulting in application's that cannot reason about failure,
- lack of control mechanisms that allow for the persistence of compound objects,
- inability to include legacy objects within a compound persistent object,
- weak association between the PID and its associated object,
- the ability to circumvent the encapsulation of an object through the protocol interface, and
- the promotion of proprietary architectures.

We advocate the superior model of *orthogonal persistence* and demonstrate how the persistent operating system *Grasshopper* can support the orthogonal persistence and *resilience* of CORBA objects. This approach to persistence in CORBA could be extended to all ORBs if supported by a suitable persistent infrastructure. We contend that such an approach would aid reasoning about the semantics of applications constructed using CORBA and therefore make the design and programming of distributed applications easier and less error prone.

## *9 Acknowledgements*

This work is supported by ESRC research grant number H519255403, "Integrated Operating System Support for Large Heterogeneous Archives". We would like to thank Iona Technologies for their support in particular Fiona Hayes. This work benefits from SunSoft's public domain IIOP engine [15].

## *References*

[1] R.di Bona, Anders Lindstrom, Alan Dearle, Stephen Norris, John Rosenberg, Francis Vaughen. "Persistence in the Grasshopper Kernel", in *Proceedings of Proceedings of the 18th Australasian Computer Science Conference*, pp. 329-338, 1995.
http://www.gh.cs.su.oz.au/Grasshopper/Papers/Papers.html

[2] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison, "An Approach to Persistent Programming", *The Computer Journal*, **26**(4), pp. 360-365, http://www-fide.dcs.st-and.ac.uk/Info/Papers1.html#approach.persistence, 1983.

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

[4] S. Baker, "CORBA Distribute Objects Using Orbix", Addison-Wesley, Harlow, England, 1997.

[5] A. Dearle, J. Rosenberg, F.A. Henskens, F.A. Vaughan, and K.J. Maciunas. "An Examination of Operating System Support for Persistent Object Systems", in *Proceedings of 25th Hawaii International Conference on System Sciences*, Poipu Beach, Kauaii, pp. 779-789, 1992.

*URL* http://persistence.cs.stir.ac.uk/~al/abstracts.html#GH-09.

[6] R. Fabry, "Capability-Based Addressing", *Communications of the ACM*, **17**(7), pp. 403-412, 1974.

[7] C. Fidge. "Timestamps in Message-Passing Systems That Preserve Partial Ordering", in *Proceedings of 11th Australian Computer Science Conference*, pp. 56-66, 1988.

[8] D. Hulse, "Store Architecture in a Persistent Operating System", Computer Science Department, University of Adelaide, Adelaide, 1998. ftp://persistence.cs.stir.ac.uk/pub/papers/dave-thesis.pdf

[9] M. Atkinson and R. Morrison, "Orthogonally Persistent Object Systems", The *VLDB Journal*, **4**(3), pp. 319-401, 1995.

[10] OMG, "The Common Object Request Broker: Architecture and Specification", 2.1, August 1997 1995.

[11] OMG, "CORBAservices: Common Object Services Specification", July 1997 March 31, 1995.

[12] OMG, *Externalization Service Specification*, in *CORBAservices:Common Object Services Specification*. March 31, 1995. p. 8-1 to 8-32.

[13] OMG, *Persistent Object Service Specification*, in *CORBAservices:Common Object Services Specification*. March 31, 1995. p. 5-1 to 5-44.

[14] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *CACM*, **39**(9), pp. 62-69, 1996.

*URL* http://persistence.cs.stir.ac.uk/~al/abstracts.html#CACM.

[15] Sunsoft, *Inter-ORB engine*, 1995. ftp://ftp.omg.org/pub/interop.