Persistent Operating System Support for Persistent CORBA Objects

Adrian O'Lenskie, Alan Dearle and David Hulse

Department of Computing Science
University of Stirling
Scotland

Email: {aol,al,dave}@cs.stir.ac.uk

Abstract

The Object Management Group (OMG) have defined mechanisms to support the persistence of CORBA objects with both the now deprecated Persistent Object Service (POS) and the new Persistent State Service (PSS). In this paper we describe both specifications and show that they are deficient in a number of areas. We introduce an alternative CORBA persistence mechanism that makes use of a novel persistent operating system called Grasshopper. Persistent CORBA objects hosted by Grasshopper do not suffer from the problems associated with the mechanisms defined by OMG. This paper briefly describes the Grasshopper system and how it may be used to host persistent CORBA objects.

1 Introduction

The OMG have defined the Common Object Request Broker Architecture (CORBA) [8] specification to allow networked objects to communicate across heterogeneous environments. The specification has been successful in addressing environments with differing CPU architectures, operating systems and language implementations, providing a common computational model. However, OMG have been less successful in providing the same support to systems using differing persistence mechanisms, both through the now deprecated Persistent Object Service (POS) [10] and the new Persistent State Service (PSS) [11]. In this paper we show the deficiencies of both specifications, and argue for the superior model of *orthogonal persistence* [13]. We describe how we have implemented the CORBA architecture within Grasshopper [6], an operating system that provides orthogonal persistence as a basic abstraction, and compare it with the persistence mechanisms defined by OMG.

The paper is structured as follows: Section 2 introduces the CORBA Architecture. Section 3 shows the motivation behind the current Persistent State Service (PSS) by examining the recently deprecated Persistent Object Service and highlighting its failings. In section 4 the Object by Value specification, which is fundamental to the PSS is described, before the PSS itself is examined. Section 5 describes the motivations for orthogonal persistence. The application of orthogonal persistence in a persistent operating system and how it may be applied in a CORBA environment is the subject of Section 6 before section 7 concludes.

2 The Common Object Request Broker Architecture (CORBA)

The OMG have defined the Common Object Request Broker Architecture (CORBA) as a standard for distributed object computing systems. The standard allows networked objects to communicate across heterogeneous environments. The architecture is divided into three sections:

- a unifying substrate (the Object Request Broker (ORB)),
- a set of standards defining application centric functionality (the CORBAfacilities),
- and a set of standards defining distribution specific functionality (the CORBAservices),

2.1 The Object Request Broker (the ORB)

The ORB provides the functionality required to enable communication between network objects. The three components that constitute an ORB 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). 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 [8].

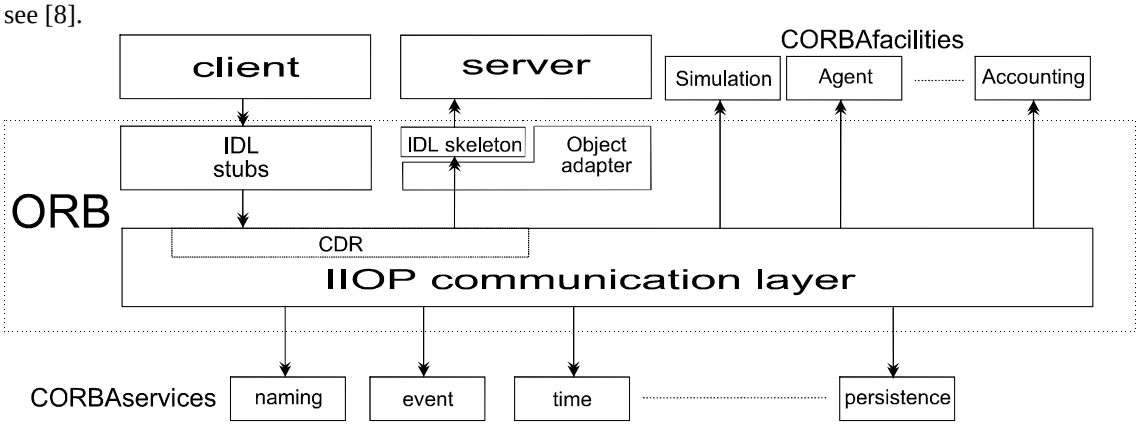

Figure 1. The CORBA Architecture

2.2 CORBAfacilities

The CORBAfacilities are geared toward providing functionality to distributed object systems from a particular domain. For example, the Text Checking User Support Facility provides the ability to:

- Initialise and free text checking objects,
- send and receive text strings with both the original text and the changed text (to support versioning in the application),
- provide the customisation of the service to be executed against the string, including at least: spelling check; hyphenation check; thesaurus; and grammar check,
- and provide the capability to query the dictionary in use, and allow alternative, application-provided dictionaries.

2.3 CORBAservices

CORBAservices are motivated by the fact that implementers of most distributed applications face common problems associated with distribution. Rather than solving the problem separately in each application, the

CORBAservices, if present, offer functionality to address these problems in applications. 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. The services include:

- Naming Service,
- Event Service,
- Concurrency Control Service,
- and the Persistent State Service.

Of particular interest is the *persistent state service* (PSS) [11], which is discussed later in this paper.

2.4 Support for Persistence

The CORBA model has been successful in bridging the gap between heterogeneous computing environments by abstracting over programming languages, operating systems and hardware technologies. However, the support provided for datastore technologies has largely been ineffectual. Figure 2a shows a typical CORBA servant[1]. The servant makes requests of the ORB (or provides services to the ORB); these requests/services are orthogonal to the local computational environment i.e. the servant is isolated from the local environment. For persistence, however, the CORBA servant must make use of non-CORBA mechanisms (for example, it may save its state to a local file store). This has the effect of creating a dependency between the servant and the local execution environment, thus anchoring it to a machine specific technology. The ideal situation is depicted in figure 2b. The ORB mediates the binding between CORBA objects and datastore; this allows the servant to be independent of all machine resources, adding power to the CORBA object model.

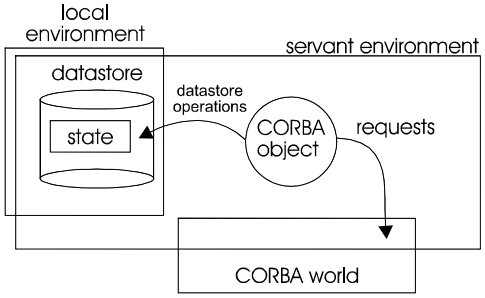

Figure 2a. Current Persistence Behaviour

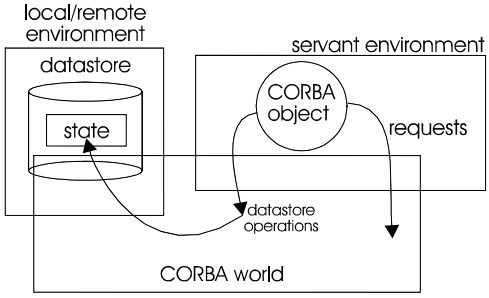

Figure 2b. Ideal Persistence Behaviour

3 The Persistent Object Service (POS)

The POS was OMG's first attempt to provide a comprehensive specification for persistence that would cater for all possible uses. This section describes the specification and details its major failings.
The motivation behind the POS was "to provide common interfaces to the mechanisms used for retrieving and managing the persistent state of (CORBA) objects" [10]. The specification indicates that the service was intended to be used in conjunction with other services. For example, a distributed database could 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 was composed of the following (independent) abstractions that combined to provide a service:

- Persistent Identifier (PID) – identifies the location of an object's persistent data in a datastore,
- Persistent Object (PO) – a CORBA object that supports an interface allowing a client to control the persistence of its state,

[1] A servant is a programming language specific implementation of a CORBA interface.

- 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 was exceedingly abstract in places thereby allowing implementations to take almost any form. In other places it was rigid and restrictive. This will become apparent after a description of how the service was expected to operate.

3.1 Summary of Interaction Between Modules

Figure 3 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 3, this process starts with the creation of a PID (Figure 3(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. In this case this occurs with a POfactory object creating an instance of a PO. The PID is passed to the POfactory along with an indication of which POM the PO is to use (Figure 3(2)).

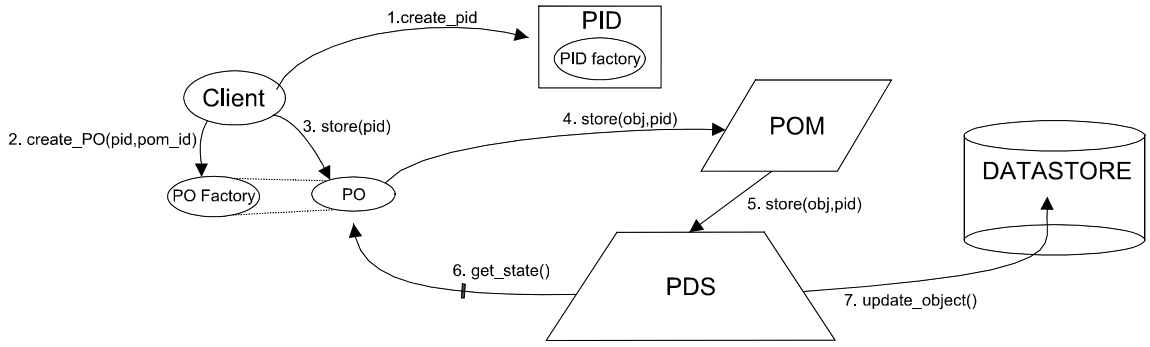

Figure 3. The deprecated CORBA Persistent Object Service

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 3(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 3(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 3(5)). The PDS extracts the state from the PO (Figure 3(6)) using the defined protocol, and stores the state in the datastore using some unspecified interface (Figure 3(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.

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)).

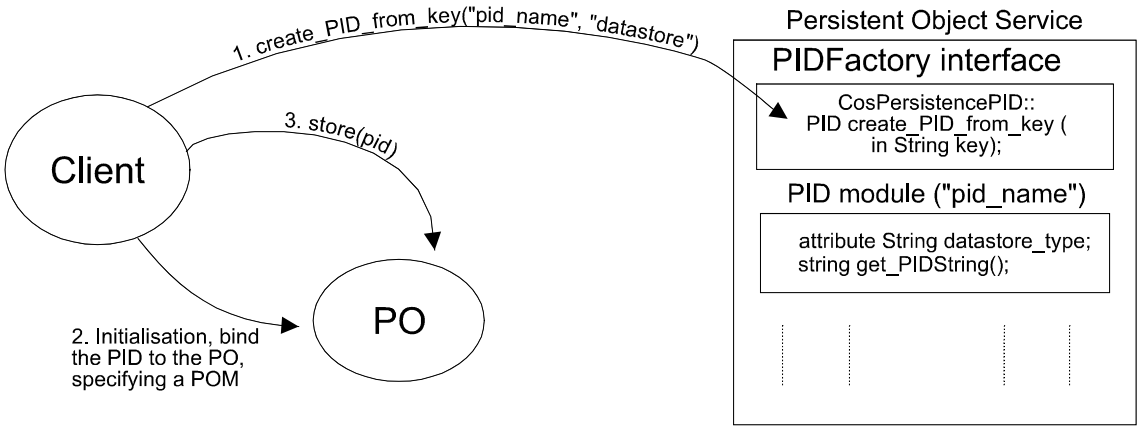

Figure 4. The client's view of the Persistent Object Service

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.

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.

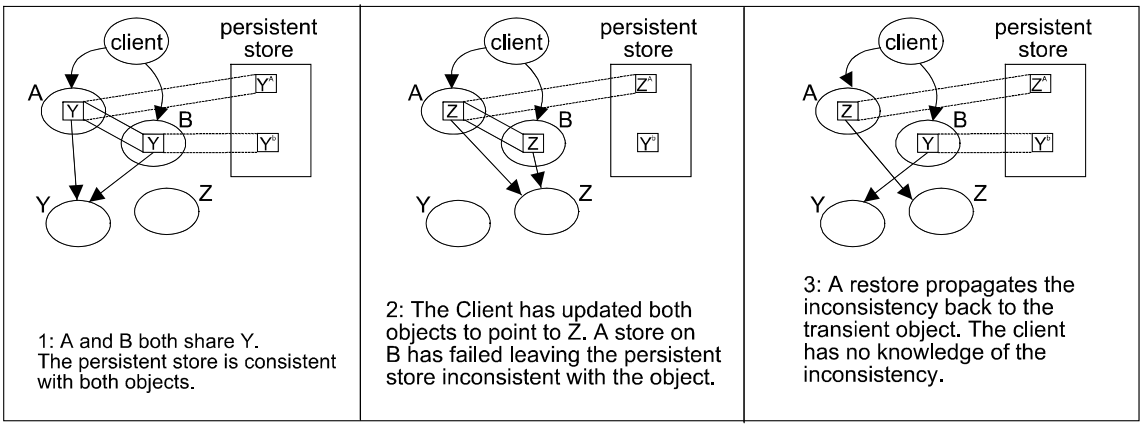

Figure 5. Semantic inconsistencies

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) [9] 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.

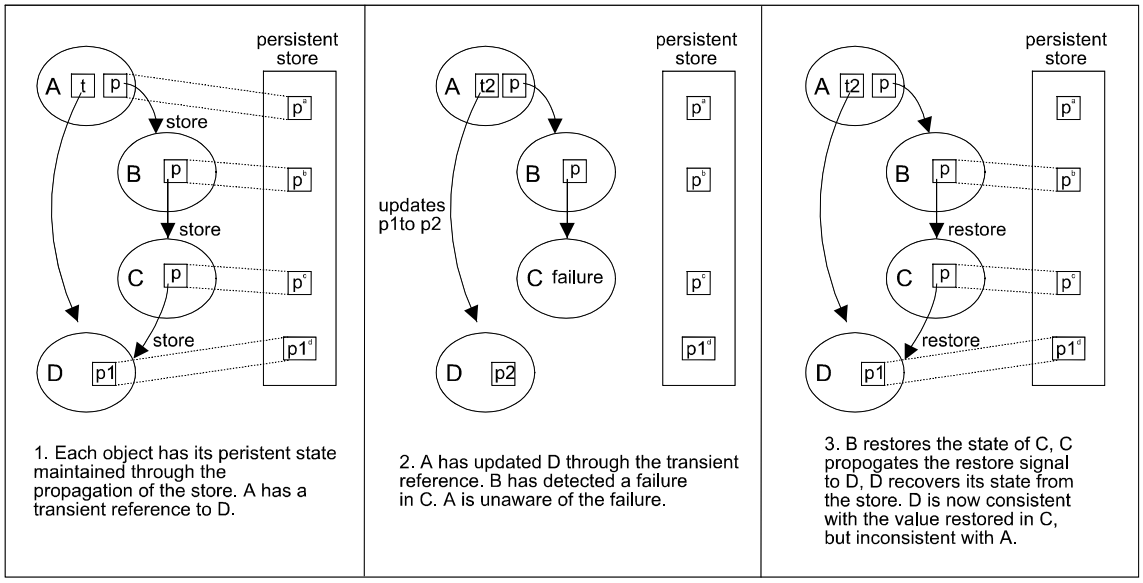

Figure 6. Lack of global control

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 (i.e. a *callback* mechanism). 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.

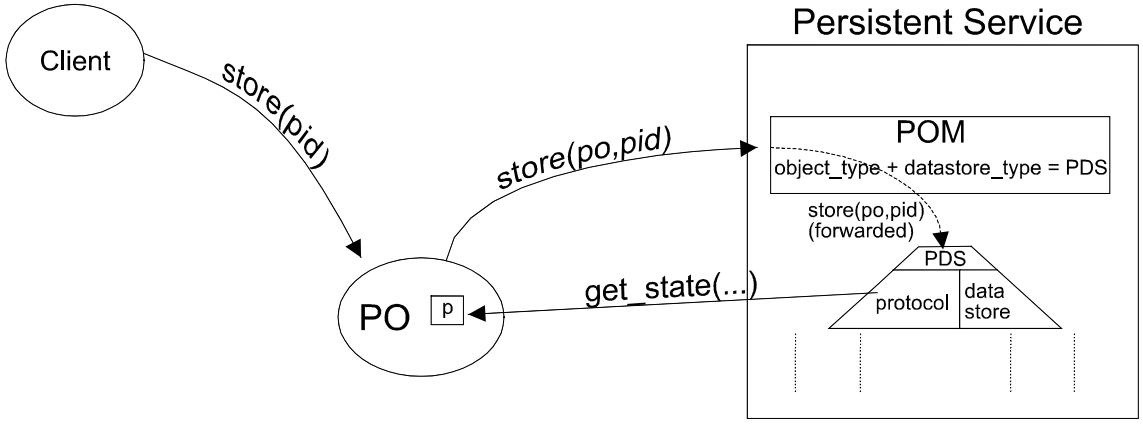

Figure 7. Persistent Object's View of the Persistent Object Service

3.3.1 Legacy Objects

The server object (the PO) must support 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 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*.

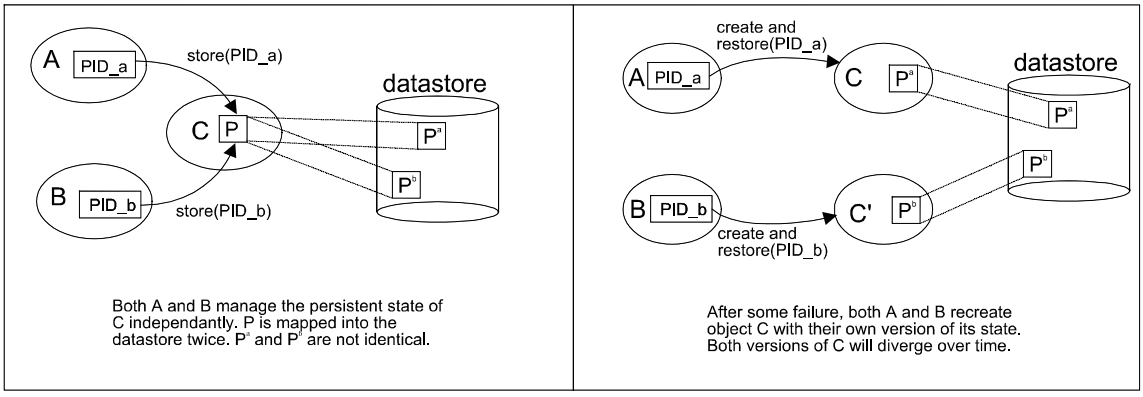

Figure 8. Weak binding between PID and object

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 a simple protocol to transfer state to the datastore, 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. This has resulted in few implementations of the specification by vendors and prompted the OMG to deprecate the specification and make a second attempt with the Persistent State Service (PSS).

4 The Persistent State Service (PSS)

The PSS is a radical deviation from the POS. Rather than attempt to encapsulate all possible uses of persistence, the PSS defines an internal interface to persistence mechanisms. This section begins by highlighting the major differences between the POS and PSS specifications before the Object By Value (OBV) specification is discussed (the PSS is built on top of the OBV specification). A more in depth study of the PSS is then presented and we describe how the PSS addresses the problems associated with the POS introduced in Section 3 before finally highlighting the failings of the PSS.

4.1 Difference between the POS and the PSS

The major difference between the POS and the PSS are:

- In the POS the persistence behaviour of an object was exposed to clients. The PSS indicates that the persistence behaviour of an object occurs totally within the domain of that object. The role clients play (if any) in the persistence process is exclusively at the candidate objects discretion (i.e. the persistence mechanisms are not visible to clients). An implication of this is the client cannot arbitrarily use persistence mechanisms without first becoming a CORBA servant.
- The POS attempted to define a specification that allowed a graph of CORBA objects to synchronise their persistence operations with each other, the unit of persistence being the graph of objects. The PSS, however, takes a more pragmatic view by employing some defined state of the object itself as the unit of persistence. The defined persistence mechanisms do not operate over graphs of CORBA objects.
- The mechanism defined to move state to/from the candidate object is no longer purely abstract (as was the case with the Persistent Data Service, Protocol and Datastore abstractions of the POS). The PSS is dependent on the use of the Objects By Value (OBV) specification to move state between the object and store.

4.2 Objects By Value

Until recently CORBA (2.1) only supported object reference semantics. This allowed networked objects to be perceived purely in terms of IDL interfaces. As all operations on any object is performed in terms of a reference, there is no explicit notion of the location of objects to the application programmer (this is especially true if all operations are treated as remote – any potential network errors are factored out). The primary aim of CORBA middleware has been achieved; distribution has become transparent to the system developer.
However there are instances where the ability to pass an actual copy of an object, rather than a reference is advantageous. For this purpose, the OMG has defined the “Object By Value” (OBV) specification to allow

pass by object semantics. For example, figure 9 shows a diary server within some company. The BigBoss object has the ability to enter appointments in the diary of any of his underlings. Each morning, the underlings will retrieve their diary to obtain their orders for the day. The underlings also would like the ability to manage their own diaries, however, for audit purposes, the orders from the boss have to be preserved on the server. With traditional pass-by-reference semantics, the server must maintain two diary objects, the original, and one the underling can alter as they please. A call to *get_diary(…)* returns a reference to a diary copy residing on the server. Underlings update their diaries with remote operations on this reference. This is a server heavy model.

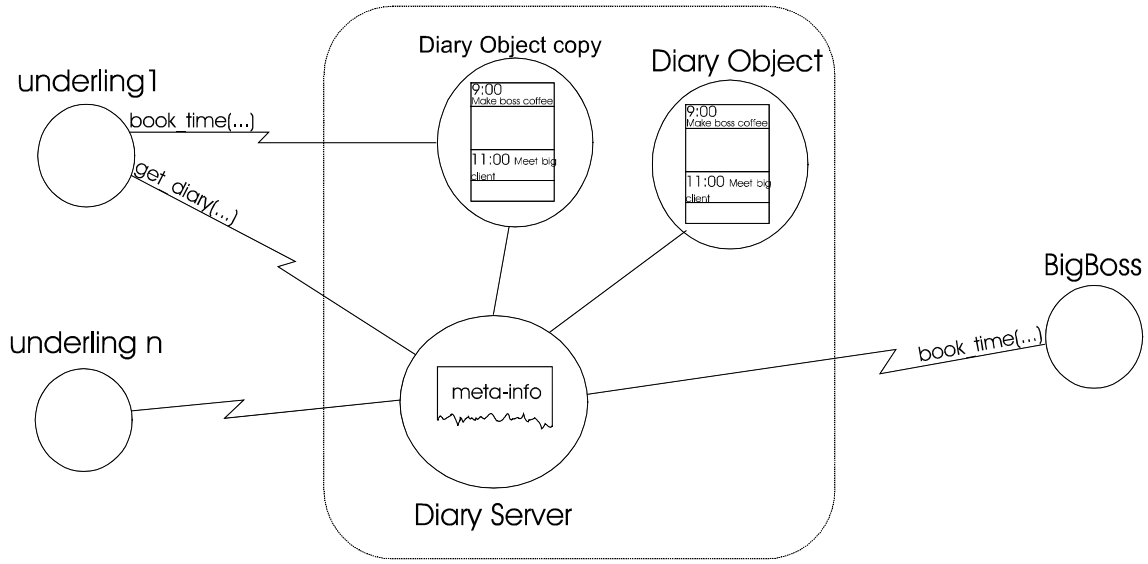

Figure 9. A traditional CORBA diary application.

An alternative solution would be for the underlings to maintain their diaries locally (figure 10). Now a call to *get_diary(…)* returns an instance of the diary object. The underling is free to change the copy as they wish, the original diary object is still maintained on the server.

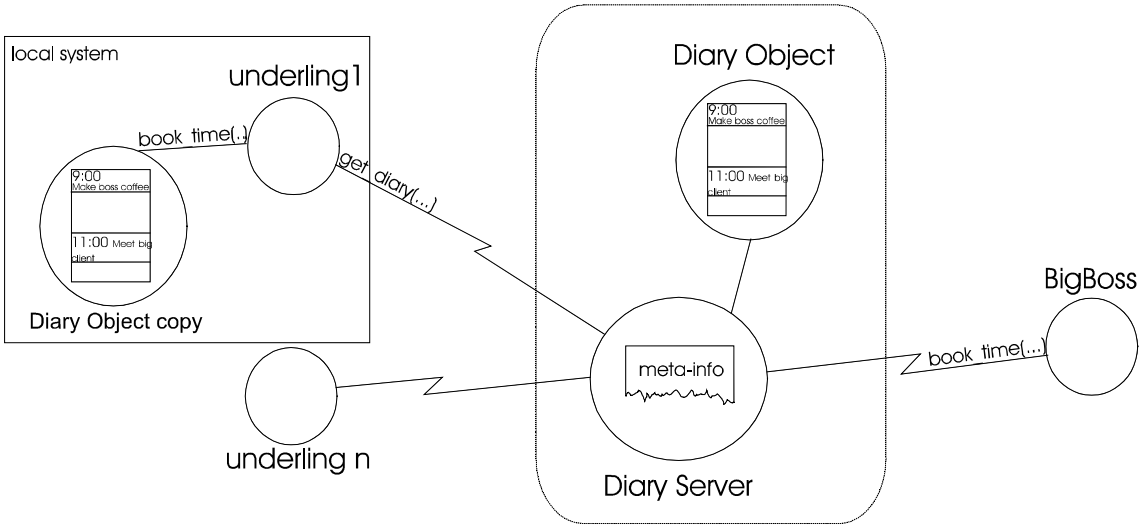

Figure 10. Object By Value diary application.

4.2.1 Object By Value properties

Although the Object By Value model increases the flexibility afforded to distributed system designers, it reduces the transparency of the CORBA model. Further, objects are no longer perceived in terms of a

public interface, this interface is compounded with the definition of state: the CORBA IDL has been extended to include the **value** type that defines this state.
Value type implementations are always local. That is, they are not CORBA objects that are registered with an ORB, but local programming language entities (for example an instance of a C++ class). However, if a value type supports an interface type, it can support CORBA object reference semantics, as long as it has been registered with the ORB.
When an object is passed by value, its state is marshalled, the receiving context creates an instance of the object and the state is unmarshalled into this new instance.
There are three possible situations that can occur when a value type is passed from a sending context to a receiving context:

- The receiving context could support the implementation directly
- The receiving context does not support the implementation directly, but can upload an appropriate implementation (possible from the sending context)
- The receiving context does not support and has no way of obtaining the implementation.

These three scenarios are depicted in figure 11. The programming language used in the implementation will affect the actual mechanism employed. For example, most languages could support the implementation directly (figure 11(1)), through the use of object factories. Figure 11(2) will be easiest implemented in languages with intrinsic support for code mobility (for example Java). If mobility cannot be supported, Figure 11(3) will be the result.

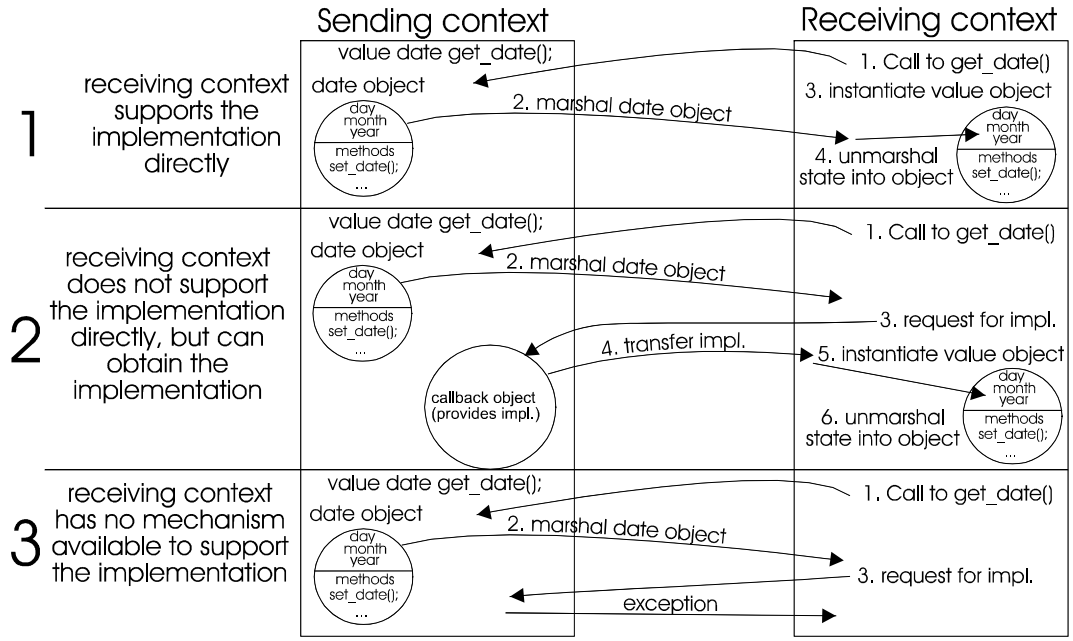

Figure 11. Three scenarios for Object By Value

4.2.2 Equivalence

The Objects By Value specification uses type structure equivalence to determine whether two object implementations are equal (more specifically, if the receiving context can support the implementation of the object directly). The structure of the IDL source is parsed to produce a 64-bit signature (using a hashing algorithm operating over the types within the source). If the receiving context expects a different signature from that received, it can deduce that its definition of the object has a different type structure to that of the sending context. In this case, it can either fail the operation or attempt to fulfil the request using information from the interface repository. If the signature is the same for sending and receiving contexts, it is highly probably that the two objects are type-structure equivalent. There is a potential problem with this definition of equivalence. Consider the two IDL code segments which are type structure equivalent below.

value AmountOfPlutonium{ **value AmountOfPlutonium{**

int killos;	**int pounds;**
int grams;	**int ounces;**
}	**}**

Although both these code segments would result in the same signature being generated by the hashing algorithm (and therefore be taken as "identical" by implementations), it can be seen that they are clearly different.

4.3 Example Implementation of the PSS

This section presents a brief example of a hypothetical PSS implementation. Although the specification is abstract enough to allow many compliant implementations, the issues raised here are likely to be pertinent in all cases.

```
value Person{
String name;
sequence <grades> gradelist;
…
};
interface pupildb{
Person get_pupil(in String name);
void set_pupil(in Person p);
…
};
```

Figure 12. Example internal IDL

Figure 12 shows an example of some internal IDL that is used to support the persistence of a CORBA value (Person). Figure 13 demonstrates how the actual persistence mechanisms (relating to the above example) might work.

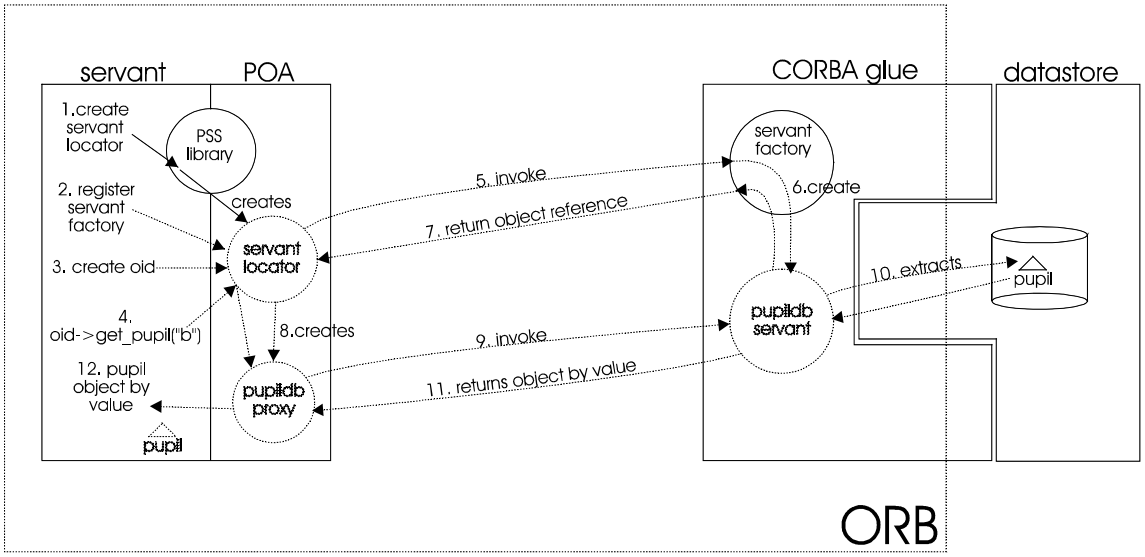

Figure 13. Example Implementation of the PSS

Initially, there is a servant supported by a PSS library and a POA (shown in solid lines on the left). On the remote (datastore) side, some mechanism (a servant factory) for creating a "pupildb" servant exists. Before the servant can make use of the persistence mechanisms, it must have the ability to bind to the remote servant factory. This is achieved by using a **servant locator**. Servant locators are described in more detail

in [12], but the basic idea is if the POA cannot service a request (because a target servant is not active, for example) a *pre-invoke* method of the servant locator is invoked. This allows the POA the opportunity to perform some activity (for example creation/activation of the target servant) that would allow the original request to be serviced.
The sequence of actions that occurs when a the servant wishes to make use of the PSS is as follows:

1. Create the servant locator. This will mediate requests for operations of type **pupildb**.
2. Register the servant factory. The servant locator will require the reference of the factory object it will invoke on operations of type **pupildb**. This reference could be obtained from the COS Naming service.
3. Create an object id. The servant requires this to perform operations on the remote datastore. An interesting point is the corresponding pupildb object does not exist at this time. The object id merely identifies a need for a pupildb servant.
4. The servant locator (the object itself does not exist) will handle an operation on this object id.
5. The servant locator invokes a "create_pupildb" operation on the remote servant factory associated with the object in 2.
6. The servant factory creates the pupildb servant.
7. The servant locator receives the actual object reference corresponding to the newly created pupildb servant.
8. The reference is bound to a local proxy of the pupildb servant, which in turn handles the original "get_pupil" request.
9. The pupildb servant is invoked with the get_pupil request.
10. The pupildb servant extracts the appropriate state from the database and places it into a value instance of type person.
11. The created value instance is returned using the object by value mechanism.
12. The servant receives the person object by value.

The specification does not define the lifetime of the servant that interacts with the datastore. The "post_invoke" action of the servant locator might delete the remote servant after every operation (which allows for easier database coherency control at a cost of efficiency) or it may allow the servant object to be cached.

The initial insertion of state in the datastore is performed by **persistent value** factories. These factories are responsible for creating a unique (within the address space) identifier for this state. The PSS supports this through the specification of two abstractions. The first is the **datastore handle** that identifies a specific datastore (and possibly a course location within the datastore, for example, a filename within a file system, or a table name within a relational database). The second abstraction is the **persistent value identifier**. This identifies a particular instance of the persistent value within the datastore. It may be an offset within a file for a file system, or could be the primary key in a relational database.

In our example (Figure 13) the datastore handle is maintained as part of the "CORBA glue" to the datastore. In this case, the persistent value identifier will be the name of the pupil being accessed/updated (assuming the name can be a primary key).

4.4 Comparison between PSS and POS

This section shows how the PSS addresses the problems identified with the POS (in section 3) and summarises the failings of the PSS.

4.4.1 No failure Model

With the POS specification, a client was the instigator of persistence operations performed by some server. However, there was no defined method of reporting error status back to this client resulting in the inability to use transaction control mechanisms to co-ordinate the persistence behaviour of graphs of CORBA objects. The PSS restricts the view of a server's persistence operations to within the server itself. Clients are unaware of any persistence mechanisms the server might employ, unless there is some explicit application level agreement to suggest otherwise. Further, a failure model is specified.

4.4.2 Local Persistence Operations Yield Global Inconsistencies

The POS was designed to allow persistence operations to flow through a graph of CORBA objects. A client that is dependent on a graph of CORBA objects has no knowledge of persistence operations being performed by nodes in this graph. The PSS avoids this problem through the redefinition of the unit of persistence. Rather than supporting the asynchronous propagation of persistence operations on a graph of CORBA objects, the PSS supports value types (not CORBA objects) which may be recursively defined or which may involve arbitrary graphs of value types. From the perspective of the application developer, a persistence operation on a value type occurs atomically; therefore behaviour is easier to reason about. Although it is not explicitly stated in the specification, if there is a requirement for a graph of CORBA objects to be consistent with each other, then the application developer must use other mechanisms (for example, those supplied by the Object Transaction Service [9]) built into objects at the application level.

4.4.3 Legacy Objects

The POS could not support legacy objects due to the object being required to support a protocol. Likewise, the PSS cannot support legacy objects due to the candidate server being able to support the Objects By Value specification.

4.4.4 Loose Association Between PID and Object

The POS defined a weak association between a persistent identifier and the object it denoted. That is, an object could have multiple PIDs bound to it. There was no idea of relationship between these PIDs – thus the POS could not support the persistence of arbitrary graphs without risk of a loss of referential integrity. The binding between the state of a CORBA object and its persistent manifestation is far tighter. The pid – datastore handle combination is guaranteed to be unique for a persistent value incarnation. This gives persistent data a strong sense of identity, which it lacked with the POS.

4.4.5 Broken Encapsulation

The POS specified that (server) objects could export a **protocol** interface to allow its state to be saved and restored. This public interface would have been a serious breach of security, allowing clients to arbitrarily alter the object's state. The PSS defines an internal interface to persistence. Clients are totally oblivious to the persistence behaviour of a server.

4.4.6 Proprietary Architecture

The POS defined a service that was composed of several key modules. The aim was to allow modules from different vendors to be combined to provide the POS implementation. However, due to the nature of the interaction between modules, and the lack of specification for this interaction, it would be highly unlikely that two different implementations could be able to interact. The PSS model is far simpler. By mapping the state of an object onto an intermediate form (the value type), flexibility is gained. New datastore types can be supported through the provision of the mapping from the intermediate format to the new datastore.

4.5 Overview of the PSS

The PSS defines the mechanism necessary to map the transient data of objects onto a datastore (figure 14). The state is converted into an intermediate, datastore neutral, form (using the object by value specification). The intermediate form is unmarshalled to its initial structure within a servant. This servant maps the data onto a particular datastore. Different datastore technologies are accommodated by producing appropriate datastore mappings (i.e. different datastore specific servants). The Objects By Value specification is extended to provide the interface to persistence using **persistent values**. **Servant locators** are defined to allow a CORBA object to be associated with a persistent value, and a three-level transactional model is proposed (non-transactional data access, local transactional access and global transaction control mediated by the Object Transaction Service).

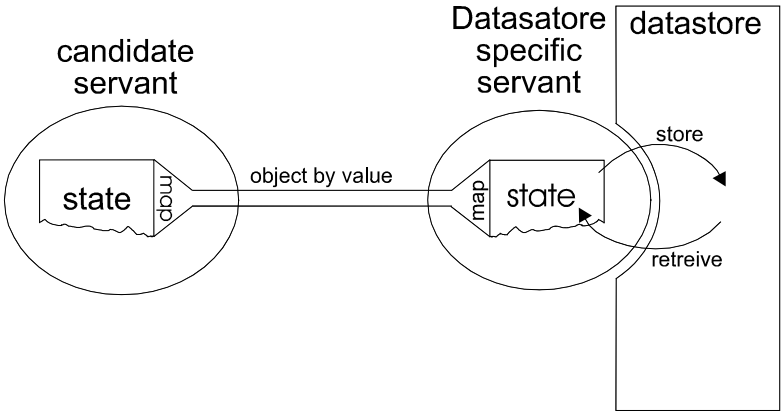

Figure 14. Overview of the PSS

4.6 PSS Failures

The PSS is designed to bridge the gap between the worlds of CORBA objects and datastore technologies. However, the worlds could hardly be further apart. On one side the emphasis is firmly on openness and inter-working, while on the other proprietary architectures prevail. This has the effect of making the object less “reusable” on systems with different database technologies. The PSS attempts to solve this by placing an abstract layer between the CORBA object and datastore. Based on Objects By Value, this layer requires two extra transformations (servant to OBV, and OBV to datastore specific servant) before the data is mapped into the datastore. This can only be detrimental to the performance of the application.

Although the specification suggests that datastore specific code could be automatically generated, tools to provide such a task would be bound to a particular datastore. These tools would also depend on the datastore having a well-defined schema. For example, it is hard to see how tools could be written to produce the database specific code supporting a legacy flat file system. A range of different (CORBA and non-CORBA) applications potentially share this data; agreement regarding the structure of the data is necessarily ad hoc.

Although the specification defines how the lifetime of the persistent state is managed (using persistent value factories); the mechanisms it specifies (datastore handle and persistent id combination) for identifying this state are abstract enough to prevent the integration of rival implementations. For example, our example implementation (figure 13) maintains the database_handle abstraction within the database specific implementation. Another compliant implementation might choose to maintain the database handle on the originating servant (imagine if some state was required to be mapped into multiple datastores).

The Object By Value specification is also fundamentally flawed. Value object equivalence is determined on type structure of that object (two object values are probabilistically the same type of object if their type structures are identical). The implication of this is there is ambiguity in the equivalence operation.

Related to this is a more general problem of object evolution. As there are potentially multiple systems connecting to a datastore, each of these systems (and the datastore system) evolve independently. There are no semantics defined by the specification to control this evolution. Consequently, a change in the datastore will require all PSS implementations to re-evaluate the binding between datastore state and CORBA object. This is likely to involve at least the re-linking of servants using the datastore, and the regeneration of datastore specific code. The lifetime of a servant using the PSS is restricted by the evolution of the datastore.

4.7 Summary of PSS

The POS is largely unimplemented due to problems identified in section 3 of this document. The PSS is a much simpler specification that attempts to provide an abstract interface to persistent technologies. The abstraction is enabled through the use of the Objects By Value specification which maps state onto an intermediate form. The flexibility afforded by such a model is at the cost of performance. There are at least three transformations required on some state as it is mapped to/from a datastore. Although a sufficiently

powerful IDL compiler could help the application developer by producing the necessary code to map state to and from the OBV format, it is doubtful that the implementation of the PSS servants could be generated automatically. Further, as a servant evolves, the PSS servant that provides its interface to the persistent store must also evolve. This necessarily binds the application implementation code to the datastore specific code.
By large, the problems associated with the POS, and the problems introduced above are the result of some entity applying persistence operations to transient state. 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 [1]. That is, the persistence of an object should be *orthogonal* to *use*, *type* and *identification* of the object.

5 Orthogonal Persistence

The three basic principles behind orthogonal persistence [7] 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 [2].

5.1 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* that provides control over access to Grasshopper entities. After briefly describing these three abstractions, we show how they are used to implement symbolic naming, before giving an overview of the recovery mechanisms.

5.2 Containers

Containers are the only storage abstraction provided by Grasshopper; they are persistent entities that 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 that 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.

5.3 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 overlaid 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.

5.4 Capabilities

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* that are protected by segregation. For a full description of capabilities see [3].

5.5 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 that permits loci to invoke them in order to make bindings, iterate over their contents, and extract capabilities and attributes.

5.6 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 [4].
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 CORBA in an Orthogonally Persistent Environment

Both the deprecated POS and the current PSS are attempts by OMG to allow the Common Object Request Broker Architecture to abstract over current datastore technologies. Our interests lie in examining the support an orthogonally persistent computational environment can offer to CORBA systems.

In this section we discuss the architecture of the Grasshopper CORBA system before describing in more detail the actual mechanisms used to save the state of a Grasshopper based servant. We describe the consistency model used by Grasshopper, before detailing further work we plan to undertake.

6.1 Provision of (Persistent) CORBA Objects Using Grasshopper

Figure 15 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 15(1)). This namespace is consulted whenever the arbitrator receives an incoming request.

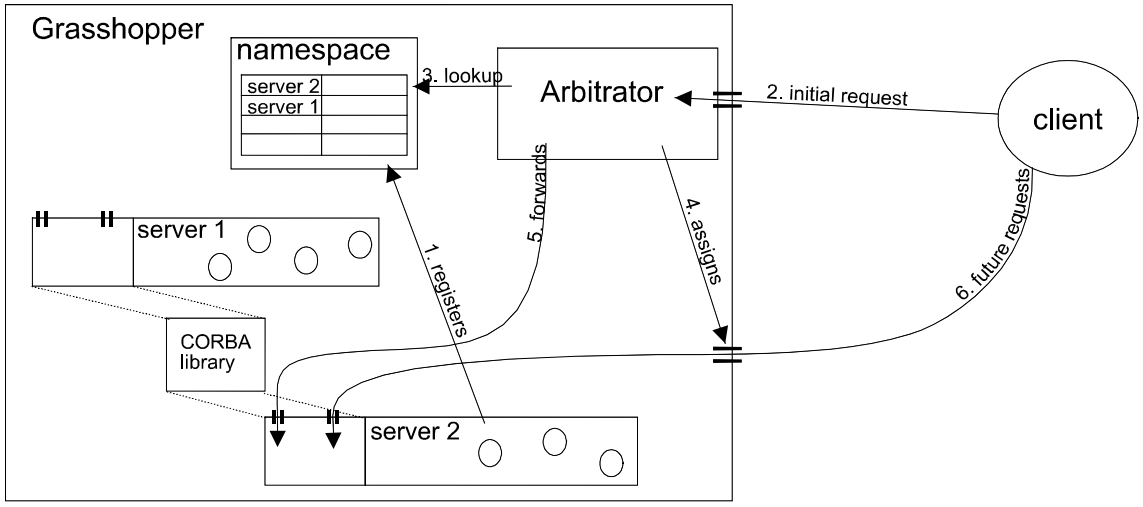

Figure 15. The Grasshopper CORBA System

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 15(2)) arriving at the arbitrator is handled as follows:

- the arbitrator resolves to which server the client is attempting to connect (Figure 15(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 15(4)),
- the server is invoked with the IIOP packet and communication port as parameters (Figure 15(5)), and
- future requests occur on the newly assigned port (Figure 15(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.

6.2 Mechanisms for Persistence

Figure 16 shows the mechanisms Grasshopper based servants employ to ensure their persistence. The procedure begins with a servant issuing a "*snapshot*" request to the snapshot server (figure 16(1)). The snapshot container determines which *containers* the *locus* requesting the snapshot has visited (here we imagine **servant a** is a client of **servant b** and has invoked some operation on **servant b**, both servants residing in separate containers). The snapshot server sends a signal to whoever manages these containers (figure 16(2)). In the case shown, both **servant a** and **servant b** are self-managing, consequently they handle their own persistence mechanisms. In this case both servants write modified data to the log-structured store and the kernel stores meta-data required to rebuild a consistent state (figure 16(3)). Finally, the log-structured storage server performs the write to stable storage.

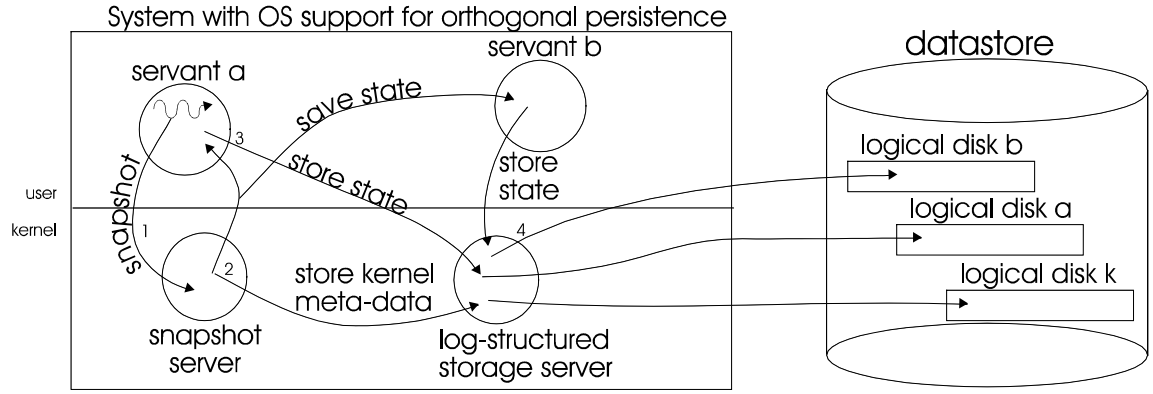

Figure 16. Mechanisms for persistence.

Unlike the PSS, this model allows the persistence mechanisms to operate over graphs of CORBA objects. This is as a direct result of the persistence mechanisms being fundamental to the operating system structure. Although the model describes servants that manage their own persistence, this could quite easily be handled by a generic manager, simplifying the implementation of servants.

6.3 Persistence of CORBA Objects in Grasshopper

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 [5]. 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.

7 Conclusions

Reasoning about the persistence of an object (especially in a distributed context) is difficult. We have shown that the POS lacked 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.

The newly defined PSS addresses most of the problems through simplification. Although this results in a simpler, less abstract model, it still suffers from major problems, namely:

- the specification of abstractions identifying persistent state could lead to proprietary architectures,
- there is a performance overhead in mapping state to the intermediate OBV form,
- no semantics are defined for controlling object evolution,
- there is a weak concept of equivalence (defined in the OBV specification), and
- there is no explicit support for graphs of CORBA objects.

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 supported at either the language or operating system level. 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.

8 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 [14] and ORL’s omniORB[15].

9 References

[1] 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, URL http://www-fide.dcs.st-and.ac.uk/Info/Papers1.html#approach.persistence, 1983.

[2] 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.

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

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

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

[6] R. di Bona, A. Lindstrom, A. Dearle, S. Norris, J. Rosenberg, and F. Vaughan. "Persistence in the Grasshopper Kernel", in *Proceedings of 18th Australasian Computer Science Conference*, pp.329-338, 1995. URL http://persistence.cs.stir.ac.uk/~al/abstracts.html#GH-09

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

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

[9] OMG, *Object Transaction Service*, in *CORBAservices:Common Object Services Specification.* March 31, 1995. p. 10-1 to 10-86.

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

[11] OMG, *Persistent State Service Specification*, in *CORBAservices:Common Object Services Specification*. May 19, 1998.

[12] OMG, *The Portable Object Adapter*, in *ORB Portability Joint Submission orbos/97-05-15*. May 20, 1997. p. 3-1 to 3-62.

[13] 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.

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

[15] ORL, omniORB. 1998. URL http://www.orl.co.uk/omniORB/omniORB.html.