Nov 24, 95A

Implementing Corba Persitence Service

Jan Kleindienst, František Plášil, Petr Tůma

Tech.Report No 117 December 1995

Charles University Prague
Department of Software Engineering
Malostranské nám. 25
118 00 Prague 1
Czech Republic

ImplementingCORBA Persistent Service[3]

Jan Kleindienst[2], František Plášil[1,2], Petr Tůma[1]

[1] *Charles University*
Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské náměstí 25, 118 00 Prague 1,
Czech Republic
phone: (42 2) 2451 0286 ext. 266
fax: (42 2) 532 742
e-mail:{plasil, tuma}@kki.ms.mff.cuni.cz

[2] *Institute of Computer Science*
Czech Academy of Sciences
Pod vodárenskou věží
18000 Prague 8
Czech Republic
phone: (42 2) 6605 3291
fax: (42 2) 858 5789
e-mail: {kleindie, plasil}@uivt.cas.cz

Abstract. The CORBA specification proposes a collection of Object Services that CORBA objects may use to enhance their functionality. In this paper, the authors share their experience gathered during the design and implementation of the Persistent Service as a part of the TOCOOS project. Moreover, the paper discusses benefits and drawbacks of reusing other Object Services, particularly Replication and Externalization, to support the Persistent Service. As the inter-dependencies among the three Object Services are rather complex and potentially circular, the paper aims at providing the reader with an analysis resulting in a set of guidelines a designer or a user of these services can benefit from when resolving inherent trade-offs.

Keywords. OMG, CORBA, IDL, CORBA object, Object Services, persistence, relationship, externalization.

[3]This work was done as a part of TOCOOS, a project funded by the COPERNICUS Program, project CP 940247; the work was also partially supported by GA ČR Grant No. 201/95/0976

1 Introduction

1.1 CORBA and TOCOOS

Around 1990, the Object Management Group (OMG) introduced the Objects Management Architecture for distributed systems [OMG90], which defines *abstract object model*. In 1991, OMG defined an industry standard called the Common Object Request Broker Architecture (CORBA), based upon a concrete object model derived from the OMA abstract object model. Among the CORBA-compliant systems (CORBA implementations) currently available from different vendors are e.g. Orbix [ORBIXa, ORBIXb], SOM [IBM93a, IBM93b], and DOME [DOM93].

The first version of the CORBA standard specification, CORBA 1.2 [OMG92], comprises many components together providing a foundation for performing transparent remote request calls from the requestors of services (*clients* or *client applications*) to the providers of services (servers). In principle, a request requires an operation to be executed upon an object (*target* or *server object*) provided by a server. The functionality of server objects is specified via the Interface Description Language (IDL) defined in [OMG91]; in addition, mapping of IDL into frequently used high level programming languages such as C an C++ is specified [OMG91, OMG93]. Moreover, an IDL compiler typically allows for a server object to be used as an ordinary object (by generating an access to a stub or a proxy in the client application); basically, the abstraction provided by accessing a server object via a stub (proxy), both specified by the same IDL interface, is referred to as *CORBA object* [OMG94a].

The CORBA 1.2 standard also proposes a collection of Object Services [OMG92], that facilitate CORBA supported objects with additional functionality such as creating and deleting new objects (Lifecycle Service), looking up a server with a specific interface in another CORBA environments (Trading Service), or managing persistent objects (Persistent Service). The functionality of an object service is specified as a set of interfaces specified in IDL, e.g. [OMG94a, OMG94b]. Typically, an object service is used by inheriting from the set of IDL interfaces specifying the object service.

It might be the case, that a particular service requested from a client is not available in the CORBA implementation the client is running on. However, it might be available in another CORBA implementation produced by a different vendor. Generally, the request format is vendor-dependent. The CORBA 2.0 standard [OMG94d] specifies the ways of interconnecting different CORBAs either by transforming requests in a bridge/gateway or by transporting requests in the standardized Internet Inter-ORB Protocol (IIOP).

In 1994, we started to participate in the TOCOOS Copernicus project (CP940247; other partners: Mari (UK), IONA Technologies (IE), CYFRONET (PL)), the goal of which is to design and implement the bridge between two CORBA implementations: Orbix and DOME [SUZ96] and also to design and implement a subset of Object Services that would furnish the bridge with enhanced functionality, such as persistence and fault-tolerance. Of the subset, the Persistent Service appears to be of the utmost importance, since it will support the bridge's key features, such as crash recovery and fault-tolerance.

1.2 The goal of the paper

The purpose of the paper is to articulate the lessons we have learned from designing and implementing the CORBA Persistence Service. Throughout the paper, we show that it is non-trivial to fulfill one of the key design principals proposed by OMG in the Requests For Proposals (e.g. [OMG95b]), which

has been referred to as the Bauhaus principle: *"Minimize duplication of functionality. Functionality should belong to the most appropriate service. Each service should build on previous services when appropriate."*

As the OMG Persistent Service specification leaves the design of the Persistent Service functionality core upon a vendor-specific protocol, one of the main goals of the paper is to analyze those important issues that have been left unresolved by the OMG specification and to report on our solution to the related problems. Apart from this, our additional goal is to analyze the prospects of reusing other Object Services, namely the Relationship Service (expressing relations among CORBA objects, [OMG94e]) and the Externalization Service (save/load CORBA object to/from a stream, [OMG94g]), in an implementation of the Persistent Service. As the inter-dependencies among the three Object Services are rather complex and potentially circular, we aim at providing the reader with an analysis resulting in a set of guidelines a designer and a user of these services can benefit from when resolving inherent trade-offs.

1.3 Structure of the paper

The paper has the following outline: In Section 2, we provide the reader with the general issues associated with the design and implementation of the Persistent Service - determining object persistence, creating persistent objects, saving, loading, updating object state, etc. Section 3 briefly summarizes the key concepts and principles introduced in the OMG Persistent Service specification. Our implementation of the Persistent Service is described in Section 4 with the emphasis to report particularly on the parts left unresolved in the OMG specification. Section 5 focuses on analyzing the potential reuse of other Object Services in an implementation of the Persistent Service; it discusses two of them in more detail: the Relationship Service and the Externalization Service. Section 6 closes the paper by summarizing the lessons we have learned from our implementation of the Persistent Service, especially while balancing necessary trade-offs, and particularly while analyzing the option of reusing other Object Services in an implementation of the Persistent Service.

2 Persistence: design issues in general

2.1 The key concepts and implementation issues

The concept of *persistence* has been specified and studied by many researchers; we refer the reader to books [Tan95, Mul94, CDK94], the proceeding of the dedicated workshop referenced in [SKW92, DdBF+92], and other publications, e.g. [SF94, AJL92, BM92, CBHS93]. In this paper, we will limit ourselves to persistence of objects in the CORBA environment. In this respect, the following are the principal concepts to which we will refer:

persistent object - an object, the lifetime of which can exceed the lifetime of the application it is used in.

persistent state (of a persistent object) - the n-tuple of values corresponding to the n object's attributes. Certain object attributes, usually with a very limited lifetime, can be considered auxiliary; their values may not be embodied in the object's persistent state.

dependencies (of a persistent object PO) - the set of all the objects targeted by a reference (either standard or defined by the Relationship Service) from the PO.

transitive closure of dependencies (of a persistent object PO) - the set of all objects reachable from PO via references transitively over all nested dependencies; this is an analogy of the "deep copy" concept [Str94].

In a design of object persistence, several implementation issues and tradeoffs must be resolved; in particular, how to activate/deactivate object persistence property, how to employ the standard language constructs of object creation to express object persistence property, how to save and load persistent state, and how to decide upon updating policy. The rest of Section 2 is devoted to an analysis of these issues.

2.2 Determining object persistence property

2.2.1 Statically

At the compilation time, some of the application objects are statically denoted as being persistent, typically by inheriting from a persistent base class. Such objects will hold their persistent property forever within the time scope of the application; there is no way they can cease being persistent at runtime.

This approach is relatively easy to implement. However, the necessity to activate the object persistent property statically and the impossibility to deactivate it later is not very convenient for the user. Thus, this black-or-white approach is not desirable.

2.2.2 Semidynamically

This approach is a modification of the static approach by a runtime enhancement: the persistent property of statically denoted objects can be *dynamically* activated and deactivated. This provides the user with the runtime ability to decide about an object’s persistence. Yet this control is limited to those classes of objects which have been *statically preselected* [Mey88], e.g. by inheriting from the *persistent object* base class..

2.2.3 Dynamically

This highly desirable approach, which is implicitly suggested by the CORBA specification of persistent service [OMG94b] (Section 3.1), allows the user to decide *dynamically* on the persistent property of all objects (*orthogonal persistence* [MA90]). Objects can activate and deactivate their persistent property an arbitrary number of times within the lifetime of the application.

2.3 Employing standard language constructs for expressing persistence property

In this paragraph, we analyze employing the standard language constructs of object creation for expressing an object’s persistence property.

2.3.1 Static object

A static persistent object is always created via a static constructor call. The constructor contains a piece of code that performs the act of persistent object creation. The drawback of the static creation is that the control of the destructor call is very limited, if not impossible. Another problem is related to the static persistent objects global in scope. A constructor of such an object is called before the

`main()` function of the program starts; in this case, the persistent service module can be not properly initialized.

2.3.2 Dynamic object

In the case of dynamic object creation, there seem to be two philosophically different approaches: either all intelligence of setting the object's persistent property hidden inside a "smart" constructor, resp. inside all the constructors of the corresponding class(es). Alternatively, all the code handling an object's persistent property is performed by an overloaded `new` operator.

As to the smart constructor approach, the creation a new persistent object takes lexicographically the same form as for creating a classical volatile memory object, e.g.

```
Persistent Object *po = new Persistent Object ();
```

The code in the smart constructor must ensure the following actions:

1. Assign a *persistent identifier* (PID) to the current `Persistent Object` instance, and
2. connect the current `Persistent Object` instance in the volatile memory with its data image allocated in the persistent memory for the purpose of later updating.

As to the overloaded `new` approach, the `new` operator can be overloaded either on "per class" or "per program" basis.

a) Per class overloading

   If there is a `new` operator overloaded *per class*, then it inherently knows the type of the class that should be created. In this case, the code of creation of a new persistent object looks lexicographically the same as for the smart-constructor approach:

```
PersistentObject *po = new PersistentObject ();
```

   Thus, the user must define a `new` operator for every class that can potentially have persistent instances.

b) Per program overloading

   Considering the case of having only one overloaded `new` operator per program (i.e. the one hiding the standard library `new` operator), the conclusion can be made that there is a need of supplying the information about the class type of the object being created explicitly as the second parameter of the `new` operator. The C++ language [Str94] offers no way of retrieving and processing the type information at runtime. Thus, the following code fragment

```
PersistentObject *po = new ("PersistentObject") PersistentObject ();
```

   is of the same meaning as the one in a).

   Looking at the code fragment, a threat of creating inconsistency in the `new` operator parameters by mistake can be perceived. Such a threat could be overcome e.g. by using a macro call.

2.4 Saving and loading object state

For saving and loading an object's persistent state, there are two principal approaches: First, by providing specialized object methods for every persistent object which serve to save and load the object's persistent state. For simplicity, we assume this to be done by two "comprehensive" methods *save_state* and *load_state* (Section 4.4.1) designed to save/load all the persistent object's state at once. Basically, the methods can be made public, and thus a part of the object IDL interface, or they can be made accessible to the underlaying persistence control only - e.g. by using the friend construct in C++. While making *save_state* and *load_state* public works nicely also for distributed calls, the later alternative is limited for the server side only.

Second, saving and loading of an persistent object's state can be done by placing the object into a persistent address space. A number of approaches towards the persistent address space architecture exist; to those known at most belong: [AJJ+92, AJL92, PSWL94, SDP93, SF94, FS94b, SG91a, SG91b, SKW92, DdBF+92, and HCF+95].

2.5 When to update the persistent state

The updating of persistent state can be application controlled or system controlled. In the former case, updating is done by explicit calls of *save_state* and *load_state* (Section 4.4.1) from the application.

When updating is system controlled, one of the following methods is used:

a) Updating is done at system well-defined moments, e.g. when deleting an object, an error/exception is raised, the end of the application is reached, etc. Basically, this approach is used in [ORBIXa, ORBIXb].

b) Updating is transaction based; it means that the updating is done at the end of the out-most level transaction. Naturally, this strategy requires the persistence implementation to cooperate with transaction support. In Orbix, this can be achieved by employing the filter concept [ORBIXa, ORBIXb].

c) True persistent virtual memory is employed; the state of an object saved in such memory is implicitly persistent and is up-to-date at all times [AJL92, HKPCS95].

2.6 What is to be saved

In addition to the issue of when to update the persistent state (Section 2.1), the question arises of *what exactly is to be saved.* The following alternatives can be identified: either just the persistent state of an object is saved (the object's dependencies are not saved), or the persistent state of an object is saved together with the persistent state of the object's dependencies. The dependencies can be saved recursively up to n-th level; in general, the transitive closure can be saved.

To implement saving of dependencies, the following techniques are used:

a) The current status of relations among objects is not evaluated; dependencies are saved implicitly by defining groups of object; such a group must be closed with respect to inter-object relations. As examples see clusters in COOL [HMA90, AJJ+92, ALJ92] or groups in SOM [IBM93a, IBM93b].

b) Full persistent state is saved with explicit evaluation of dependencies. This can be either application controlled (explicit) or system controlled (based on hardware-supported reference identification). In the later case, the techniques of early or late pointer swizzling are typically used [SKW92, VD92].

3 Persistence: the OMG approach

3.1 What OMG defines

The Persistent Service is specified in [OMG94b], in which the IDL specification of three basic interfaces are provided: *Persistent Object* (PO), *Persistent Object Manager* (POM), and *Persistent Data Service* (PDS). Fundamentally, these interfaces comprise the same methods: *connect(), disconnect(), store(), restore(),* and *delete().*

PDS supports a collection of pairs <*Datastore*, *Protocol*>. *Datastore* actually ensures saving and loading of the PO's data. Each *Datastore* object is identified (localized) by the pair <`datastore_type`, `datastore_type_instance_id`>. *Protocol* describes the way PDS transfers data into and from the PO. Both *Datastore* and *Protocol* are not standardized; however, [OMG94b] offers three examples of *Protocol* and a specification of *Datastore_CLI*, which might be used as "a uniform interface for accessing many different Datastores". Generally speaking, PDS communicates with the PO through a *Protocol*, and with the datastore via either *Datastore_CLI* or a proprietary (not defined by OMG) *Datastore* interface.

To be able to use the Persistent Service, each CORBA object must inherit from the *PO* interface. Each *PO* object supports a *Protocol* interface and can communicate with any PDS supporting that *Protocol*.

POM dynamically resolves the binding between PO and its PDS, given a PID of a PO and the *Protocol* supported by the PO. Here, PID is an identifier, uniquely denoting the object derived from PO in a datastore. Thus, PID is basically represented as a triple <`datastore_type`, `datastore_type_instance_id`, `key_to_PO`>. For example, <FS, hostname, path+offset> for a file-system-based datastore, or <DB, DBname, key> for a database-like datastore. Under the assumptions

a) POM knows about all available PDSs and the combinations of *Datastore* and *Protocol*, each PDS can support (implementation: POM keeps this information either in a configuration file, or registry, or via a dedicated interface), and

b) given a PO, POM knows which *Protocol* the PO supports (implementation: the supported *Protocol* is deduced for example from the PO type),

the POM resolves PDS in the following steps [OMG94b]:

1. get the `datastore_type` and `datastore_type_instance_id` from the PO's PID
2. get *Protocol* supported by the PO
3. localize a *Datastore* object using the pair <`datastore_type` , `datastore_type_instance_id`>
4. determine PDS given the pair <*Datastore, Protocol*>

3.2 Integrating with other services

The specification of the Persistent Service [OMG94b] discusses the integration of the Persistence Service with other Object Services. The discussion separates the services that potentially may use Persistence Service, such as Backup/Restore Service or Replication Service, and the services that may be used by the Persistent Service, such as Externalization Service or Relationship Service. In Section 5, we discuss the pros and cons of integrating the latter two services into our implementation of the Persistent Service.

4 Persistence: our design approach

4.1 Our design goals and boundaries

When deciding on the design issues of our Persistent Service implementation, we have been significantly influenced by two essential properties of the target application, the bridge. First, the bridge uses CORBA distributed objects as well as local C++ objects. The persistent service implementation must be able to handle both local and distributed objects. Second, parts of the bridge will run in different CORBA environments. The Persistent Service implementation must not depend on any CORBA-specific properties of the target environment.

The overall functionality of the service, however, should not be degraded by the decisions taken as a result of meeting the requirements mentioned above. Furthermore, the service implementation should meet the following objectives:

a) The implementation must be able to work as an independent Persistent Service in the Orbix environment.

b) The implementation should not depend on other services unless their suitable implementation is readily available.

c) The service should not put an additional burden on the application.

4.2 Our design architecture

Our Persistent Service design architecture can be divided into three parts hierarchically related with respect to their functionality. The Typed Data Part is responsible for accessing datastores; its only purpose is to provide a common interface for accessing various datastore types. The Persistent Object Part uses the underlying Typed Data Part to save the contents of individual persistent objects. The Persistent Service Part coordinates the previous two parts to provide the client with a simple-to-use Persistent Service interface (*POM* and, potentially, *PO*) .

Figure 1 depicts the structure of our implementation in terms of the OMG Persistent Service specification [OMG94b]. As indicated in this figure, the three parts of our architecture roughly correspond to the major components of the OMG specification: the Typed Data Part implements the OMG Datastore component, the Persistent Object Part contains important sections of the OMG Protocol, and the Persistent Service Part represents both the OMG Persistent Data Service and the OMG Persistent Object Manager components.

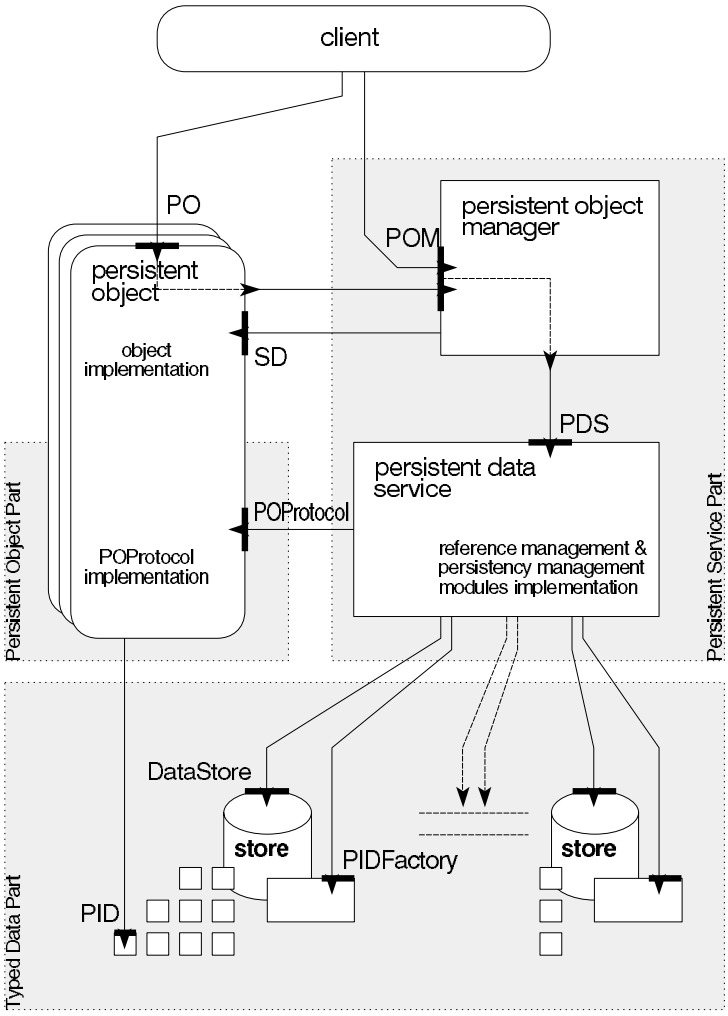

Figure 1

4.3 Typed Data Part

This part of the architecture is embodied by the Store Access Module responsible for accessing data on external storage media. Apart from the storage access primitives, the module is also capable of providing single level transactions necessary for a crash recovery support in the hierarchically higher parts. The module consists of four interfaces. The first two interfaces, called *PID* and *PIDFactory*, introduce the notion of a unique persistent identifier, as defined in [OMG94b]. The other two, called *DataStore* and *DataStoreFactory*, provide means of accessing the external store:

```
module StoreAccess {
    typedef unsigned long       ulong;
    typedef sequence <octet>    tBuffer;

    exception Internal { };
    exception InvalidMediaID { };
    exception StoreIsFull { };
```

```
    exception NoTransaction { };
    exception NestedTransaction { };

    interface PID : CosPersistencePID::PID { };
    interface PIDFactory { };
    interface DataStore { };
    interface DataStoreFactory { };
};
```

The *PID* interface masks the differences among various storage media classes, thus creating the unified abstraction of a datastore containing records addressable by PIDs. The primitives are not aware of the structure of the data being saved - at this level, each record is simply a stream of bytes of an arbitrary length. Each *PID* instance denotes a single location in a store. Apart from the *location_ID* itself, the *PID* contains two strings identifying the store class and instance. Thus, the persistent identifier is global in scope. The *PID* interface is derived from the CORBA *CosPersistencePID::PID* base interface:

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

interface PID : CosPersistencePID::PID {
    readonly attribute string media_ID;
    readonly attribute ulong location_ID;
    void rewrite (void)
        raises (Internal, StoreIsFull);
    void rewind (void)
        raises (Internal);
    void save (in tBuffer data)
        raises (Internal, StoreIsFull);
    tBuffer load (in ulong length)
        raises (Internal);
    void remove (void);
};
```

The *rewrite* and *save* methods are used to put the data into a store location denoted by the *location_ID* attribute. Rewriting a location erases any previously saved data and prepares it to be filled with subsequent *save* calls. The *rewind* and *load* methods are used to retrieve the data in a similar manner. As described in [OMG94b], *PID* instances are created by an appropriate factory (once created, the *PID* remains valid until a *PID::remove* call is issued [OMG94a]):

```
interface PIDFactory {
    PID get_root_PID ( )
        raises (Internal);
    PID create_unique_PID ( )
        raises (Internal, StoreIsFull);
};
```

As the persistent IDs are being devised by the store itself, an extra care needs to be taken to provide the client with means of obtaining the store contents. By a convention, the *get_root_PID* method returns a single *PID* denoting a special record to be used for the purposes of maintaining a directory of saved data. It is up to the client to specify a strategy for the root record usage.

An instance of the *DataStore* interface represents a datastore capable of saving and loading the persistent data:

```
interface DataStore {
    void trans_begin ( )
        raises (Internal, StoreIsFull, NestedTransaction);
    void trans_commit ( )
        raises (Internal, StoreIsFull, NoTransaction);
    void trans_abort ( )
        raises (Internal, NoTransaction);
    PIDFactory get_PIDFactory ( );
    void destroy_on_remove ( );
};
```

As the persistent service is expected to be secure with respect to system failures, a certain level of fault tolerance is required from the underlying *DataStore* as well. Thus, a simple one-level transaction mechanism is introduced - any changes made to the store contents after the *trans_begin* call take effect after the *trans_commit* call successfully returns.

A particular *DataStore* implementation is always associated with a corresponding implementation of the *PIDFactory* and *PID* interfaces. The store provides its client with means to obtain an appropriate *PIDFactory* by calling the *PIDFactory::get_PIDFactory* method. The *DataStore* instances used by the persistent service are manufactured by an instance of the *DataStoreFactory*:

```
interface DataStoreFactory {
    DataStore open_DataStore (in string media_ID)
        raises (Internal, InvalidMediaID);
};
```

Once created, a *DataStore* object remains valid until a *DataStore::remove* call is issued. Removing the store, however, does not destroy its data, unless the *DataStore::destroy_on_remove* method has been called prior to *DataStore::remove*. Usually, the *DataStore* interface will be used to save data of simple types, i.e. chars, integers, strings etc. In the first version of the implementation, a simple set of macros can provide an interface to save simple data types easily and effectively.

4.4 Persistent Object Part

4.4.1 Object Storage Module

This module is responsible for saving and loading of the persistent attributes of an individual object, together with exporting the type and object reference information. As the only entity authorized to access the persistent object’s internal state is the object itself, the services of this module are exported in a form of methods provided by each persistent object (2.4). Using the terms introduced in [OMG94b], this is a part of the proprietary protocol. The interfaces of the module are:

```
module ObjectStorage {

    typedef CosNaming::Name TID;

    interface POListItem { };
    interface POList { };
    interface POProtocol { };
};
```

In order to be able to cooperate with the persistent service, an object needs to be equipped with the *POProtocol* interface:

```
interface POProtocol {
    TID get_TID ( );
    void save_state (StoreAccess::PID pid);
    void load_state (StoreAccess::PID pid);
    get_references (out POList references);
    void set_references (inout POList references);
};
```

The implementation of the persistent data service needs to be able to identify object classes at runtime. As the C++ language does not provide the runtime type information to the user, the *POProtocol* interface exports the *get_TID* method to do this - the method returns a user-supplied identifier of the object's type. The *save_state* and *load_state* methods access the object's persistent state minus the references to other persistent objects. The *get_references*, resp. *set_references*, methods save, resp. load, the references to other persistent objects. The *set_references* method is expected to remove the processed references from the reference list passed as the inout parameter. In the class hierarchy implied by inheritance, this action is necessary in order to be able to reuse the predecessor's *get_references* and *set_references* methods.

The *POList* and *POListItem* interfaces are used to export the list of references to other persistent objects. We prefer the dynamically linked list of items over an IDL sequence, as the operation of concatenating and separating several linked lists, which is likely to be used in the *get_references* and *set_references* methods, is less time- and memory-consuming compared to the same operation performed on sequences.

4.4.2 Object Factory Module

When loading dependencies of a persistent object (Section 2.6), the Persistent Data Service implementation needs a way to create (rebuild) object instances to be filled with data from a datastore. Thus, a generic object factory is required capable of creating an object instance of the type corresponding to the given type ID. In practice, the generic factory interface is more convenient for the purpose of obtaining specific object instances if it is extended by a support for registering specific object factories.

4.5 Persistent Service Part

This part incorporates both the Persistent Data Service and the Persistent Object Manager; both specified in [OMG94b]. According to the OMG specification, the main role of the Persistent Object Manager is to dispatch a function call to the appropriate Persistent Data Service. Since we have only one Persistent Data Service instance, the interface of the Persistent Object Manager simply passes all requests through to the instance.

4.5.1 Reference Management Module

As the implementation of the persistent service is expected to work with the C++ objects as well as with CORBA objects, we can not use the CORBA object IDs to identify the object instances. We need another kind of an identifier associated with every persistent object to encode inter-object references in persistent store. Fortunately, there is no need to devise another mechanism of object identification. Each persistent object is already associated with its PID, all we have to do is to promote the PID to act as an object identifier in addition to save location identifier. In an application, we need to prevent the Persistent Service from creating several object instances using the same persistent object image in the datastore. As PID is used as a persistent object identifier, a list of <PID, object reference> pairs maintained by the reference management module can be used to check whether an object has been already loaded.

4.5.2 Persistence Management Module

This module is the heart of the persistent service functionality. It exports the following specialized version of the Persistent Data Service *CosPersistencePDS::PDS* interface [OMG94b]:

```
module PersistenceManagement {

    interface PDS {
        PDS connect (in POProtocol object, in PID pid);
        void disconnect (in POProtocol object, in PID pid);
        void store (in POProtocol object, in PID pid);
        void restore (in POProtocol object, in PID pid);
        void delete (in POProtocol object, in PID pid);
    };
};
```

As defined in [OMG94b], the *connect*, resp. *disconnect*, method turns on, resp. off, the automatic updating of the object's persistent state image in the datastore. In our implementation environment, the persistent data service has no way of detecting the moments when an object is modified. Therefore we can only update the object's image at system and transaction well-defined moments (Section 2.5).

The Persistent Data Service architecture dictates the necessity of saving and loading the entire transitive closure of an object's dependencies at once. Thus, the *save*, resp. *load*, method processes the object's dependencies recursively. When saving an object, the Persistent Data Service starts with finding a PID associated with the object (a new PID is assigned if necessary). The object's type ID is saved into a location associated with the PID, the object is then asked to save its state into the location. A list of references to the object's dependencies is retrieved using the object's *POProtocol* interface; the object's dependencies are saved recursively and a list of PIDs of the dependencies is saved into the location. The process of loading an object is inverse to the process of saving.

When issuing a *delete* call, the object is removed from the list of memory resident objects and its PID is destroyed using the *PID::remove* call. This results in the object being disassociated with its persistent state, the transient state of the object is not affected by this call.

5 Discussing reuse of other Object Services in Persistent Service

5.1 The other Object Services which potentially could support persistence

The specification of Persistent Service [OMG94b] discusses very briefly reuse of other Object Services, particularly Relationship Service, Externalization Service, Trading Services, Lifecycle Services. In this section, we will focus on the issue of reusing the Relationship Service and Externalization Service as these services, if really reused, can cover substantial subtasks of the Persistent Service implementation.

5.2 Reusing Relationship Service

5.2.1 Basic concepts of Relationship Service

The goal of the Relationship Service is to provide tools for operating upon abstractions based on entity relationship diagram concepts. According to abstractions it provides, the Relationship Service is hierarchically structured into 3 levels: base level relationship (provides *role, relationship*, means for engaging entity objects, *related objects*, in entity-relationship diagrams-like structures. The second level provides *node, graph, edge* allowing thus for creating graphs of related objects. It also provides *traversal* and *traversal criteria* for traversing these graphs and, very importantly, defining subgraphs at the runtime; a traversal criteria object can use the concept of *propagation value*. Finally, the highest level provides specific relationship *containment* and *reference*.

For brevity, we have found it useful to call *r-object* an instance of the *role, relationship,* or *node* interfaces.

5.2.2 Building Persistent Service over Relationship Service

When the Relationship service is employed, two types of inter-object references are to be considered: inter-object references expresses by (i) standard object references and (ii) via r-objects; the concept of object dependencies (Section 2.1) covers the both types of inter-object references. However, when saving/loading dependencies, the two types must be treated distinctly.

Standard object references can be treated in the way described in Section 4.5.2, i.e. the transitive closure of dependencies is built by recursive evaluation of dependencies. On the other hand, the transitive closure of dependencies based on r-objects can be built by using the standard traversal mechanism offered by the Relationship Service. To preserve the flexibility of defining subgraphs dynamically at the runtime, the traversal process should use the user-defined traversal criteria object (Section 5.2.1, [OMG94e]). In order to meet the minimum requirements of the Persistent service (dependencies, transitive closure of dependencies) the Relationship Service implementation needs to fulfill at least the service levels 1 and 2 as defined in [OMG94e].

In a general case, the two reference types can coexist in transitive closure of an object’s dependencies; this even allows more subgraphs defined by different traversal criteria objects to participate in one transitive closure. Thus, a Persistence Service implementation has to respect this flexibility in defining subgraphs.

In theory, following strictly the Bauhaus principle [OMG95b] to achieve maximum Object Service reusability, all inter-object references could be expressed by r-objects only. Even though this would unify the process of searching for transitive closure of dependencies, the overhead inherent to dereferencing only via r-objects could hardly be acceptable (Section 5.2.5). In addition, the Persistent Service needs to handle references among r-objects; to avoid infinite recursion, these references cannot be expressed by using r-objects. Therefore, the Persistent Service needs to handle standard object references anyway.

5.2.3 Making r-objects persistent

In this paragraph, we will presume both types of inter-object references coexist in the same transitive closure of an object's dependencies. How the standard object references are to be handled was described in Section 4. However, to make the transitive closure persistent, r-objects themselves have to be saved/loaded as well. Thus, the key issue this paragraph is focused on is how to save/load r-objects.

As, in principle, the set of an object's attributes belonging to its persistent state must be evaluated dynamically, saving an r-object (particularly *Role*) as an ordinary object is not possible. This fact is implied by the option to determine a subgraph using *TraversalCriteria*. Consequently, to save r-objects, a special technique is necessary. Given a *TraversalCriteria* object and a root *node*, the entire subgraph must be traversed to determine its relevant nodes and edges [OMG94e]. Only after all the r-objects representing the subgraph are found, saving of the subgraph can take place. Basically, there are four ways to do so in dependence upon the way the external form of references (PIDs) is saved:

a) PIDs are saved as parts both of the *Role* and the *Relationship* objects. Although slightly redundant, this technique closely follows the principle to externalize inter-object references as PIDs. In this case, the *POProtocol::get_references* method of a role requires a list of relevant relationship objects as an additional argument, so that the irrelevant references to relationship objects inside role objects are not externalized.

b) PIDs are saved as part of the relationship objects; at load time, references in roles are reconstructed via *Role::link* calls. This technique eliminates the redundancy inherent to a); however, it requires the *POProtocol ::get_references* method of a relationship object to call *Role::link* methods.

c) PIDs are saved as part of the role objects; at load time, references in relationships are reconstructed. This alternative is only hypothetical as it would mean saving the information, which is principally associated with relationships, inside roles.

d) PIDs are saved outside both roles and relationships objects in an implementation dependent way; at load time, references are reconstructed as a part of the relationship objects rebuilding process (e.g. supplying the information on related roles to the *create* method of *RelationshipFactory*).

In our view, the b) technique is the most elegant one, although the Externalization Service [OMG94g] uses the technique d).

5.2.4 Advantages of reusing Relationship Service

Naturally, the inherent benefit of building Persistent Service upon, among others, the Relationship Service is reusing existing CORBA code. However, the dominant advantage of employing Relationship Service in Persistent Service is the power and flexibility of graph traversing operations. In Relationship

Service, it is very easy to make changes to searching for dependencies or groups of objects simply by changing the *traversal criteria* object. For that purpose, a set of specialized *traversal criteria* objects (e.g. those traversing only certain edge types) may be predefined; furthermore, the *traversal criteria* object can be parametrized dynamically at runtime.

5.2.5 Disadvantages of reusing Relationship Service

Compared to the standard C++ dereference mechanism, using the Relationship Service may slow down the application considerably, even if local caching of relationship attributes (e.g. via smart proxies in Orbix [ORBIXb]) is used. When using the Relationship Service, dereferencing an object pointed to by a relation implies calling the *Role::get_other_related_object*. The *get_other_related_object* method requires the *Role* object it is to be invoked upon and the *Relationship* object to be traversed, thus the operation of dereferencing an object involves at least one RPC call with at least two CORBA objects being passed as an argument and a result. In fact, the overhead of evaluating dependencies by means of Relationship Service may not be critical for the Persistent Service itself, as most of the time is consumed on operating with datastore; on the other hand, forcing the client to use the Relationship Service inside its applications is hard to advocate, as it makes the application program more complex (in terms of both source code and time complexity).

5.3 Reusing Externalization

5.3.1 Basic concepts of Externalization Service

The Externalization Service supports sequential saving/loading of objects from/into the CORBA environment. The externalized objects are saved on a media in the canonical form described by [OMG94g]; thus the Externalization Service allows for easy transfer of objects between different CORBA architectures.

The Externalization Service is based on three interfaces: *Stream*, *StreamIO*, and *Streamable*. The *Stream* interface represents a sequential stream of externalized objects; it is associated with the *StreamIO* interface which provides the low-level tool for externalizing an object's attributes. The code describing how an object saves/loads its state (using the methods of *StreamIO*) is wrapped inside the methods of the *Streamable* interface the object is required to inherit to make itself externalizable:

Streamable::internalize_from_stream (StreamIO)
Streamable::externalize_to_stream (StreamIO, FactoryFinder)

The methods take as a parameter the reference to a *StreamIO* object; this mechanism provides for dynamic binding between the object and the stream that externalizes the objects's attributes.

An object uses methods of the *StreamIO* interface such as *StreamIO::write_float (float)* or *Stream:: write_string (String)* to save its attributes of simple data types into the stream represented by a *Stream* object. The *StreamIO* interface also provides the following two methods for saving dependencies referenced from the object: *StreamIO::write_object (Streamble)* and *StreamIO::write_graph (CosCompoundExternalization::Node)*. The first method is used when the relations among objects are not represented via the Relationship Service; the second method is called when the Relationship Service is used to represent relations among objects. Basically, these methods are to be called recursively to ensure externalization of the whole graph of related objects. All in all, it should be

emphasized that the *Stream* interface inherently implies sequential way of saving, resp. loading, objects to, resp. from, external media.

5.3.2 How can Externalization Service support Persistence Service

As stated in [OMG94g Section 3.1], the Externalization Service has been designed to be able to integrate with Persistent Service as a specific POS protocol. On the other hand, the only reference to the Externalization Service in the Persistent Service specification [OMG94b, Section 6.17] reads: "... the Persistent Service could use this service as a POS protocol".

In fact, being inherently sequential at a higher level of abstraction, the Externalization service can support the Persistent Service, inherently based on random access to individual objects, only in a very special case - when a Persistent Service implementation does not support fine-grained updating of parts of an externalized transitive closure of dependencies. On a lower level of abstraction, the Externalization Service interfaces *Streamable* and *StreamIO* are not necessarily limited to sequential access to externalized objects. In principle, it would be possible to implement a specialized StreamIO interface, such that the *Streamable* interface could be used to access the persistent state of an object without imposing the limit mentioned above. Although not strictly adherent to the semantics described in [OMG94g], the specialized StreamIO interface implementation could make it possible to reuse the Streamable code in a client application.

5.3.3 Advantages of reusing Externalization Service

The Externalization Service provides standardized way of saving data to the stream, resp. datastore. When used, it prevents the implementor of the Persistent Service from writing the bottom-most layer of the underlying datastore and provides clean and flexible interface to the upper layers of the Persistent Service for saving/loading of an object’s state.

5.3.4 Disadvantages of reusing Externalization Service

When the Relationship Service is used to express the dependencies (Section 2.1) of the objects to be externalized/internalized, the necessity of saving/loading the r-objects (Section 5.2.1) representing the dependency graph emerges. In the Relationship Service framework, the binding between two entity objects is expressed via a pair of *Role* objects and a *Relationship* object that connects the *Role* objects. In the graph, the position of each entity object is represented by the *Node* object that contains all the *Role* objects belonging to the entity object. When the entity object requests externalization, *StreamIO::save_graph()* is called upon the *Node* object associated with it. The *Node* object must save not only the entity object but also its *Role* objects and even itself. The *Relationship* objects are saved together at the very end of the externalization process. To support the saving/loading, all r-objects must inherit from the *Streamable* interface. Just the fact, that the methods for saving/loading in the *Streamable* interface must be written by hand for each r-object type, makes r-objects "heavyweight" with respect to the Externalization Service.

The Externalization Service does not support the structured data types directly. When using the methods of *StreamIO*, a structure in an object must be either treated as an object derived from *Streamable* and equipped with the user-written save/load code or viewed as a byte region and saved/loaded by *SteamIO::write_octet()*, resp. *StreamIO::read_octet()*.

6 Conclusion

The paper is based on our experience with implementing the CORBA Persistent Service. As the OMG Persistent Service specification leaves the design of the Persistent Service functionality core upon a vendor-specific protocol, several very important issues, such as handling of related objects and interfacing with a datastore, remain unresolved in the specification. Therefore, in Section 2, we focused on analyzing possible techniques related to these issues and particularly to the crucial trade-offs we have had to face in our implementation. Our particular solution to these issues was described in Section 4; the core functionality of our Persistent Service implementation is located in the Persistent Data Service module, with the low level support routines split between the Persistent Object and Store Access module. In Section 4, we provided the reader with relevant details.

Moreover, the option of reusing other Object Services, namely the Relationship Service and the Externalization service, in an implementation of the Persistent Service was analyzed in Section 5. As for related objects, we have found it very important to treat both the standard references and the references defined by the Relationship Service in a unified way with respect to an object's dependencies; therefore, in Section 2.1, we introduced the concept of dependencies as the set of all the objects targeted by a reference of either type from a given object. Further, we concluded that reusing the Relationship Service does not grant any significant profit to the Persistent Service implementation alone; the user, however, may benefit from the ability of the Persistent Service to understand and process the graphs defined via r-objects (Section 5.2.1) of the Relationship Service. Thus, if the Relationship Service is implemented in a CORBA, the Persistent Service implementation should support both types of inter-object references; at the same time, to avoid endless recursion, the references among r-objects must be treated in a special way in the Persistent Service. For this purpose, guidelines for making the Relationship Service's r-objects persistent were articulated in Section 5.

Being in principle sequential at a higher level of abstraction, the Externalization Service can support the Persistent Service, inherently based on random access to individual objects, only in a very special case as discussed in Section 5.3. However, as we pointed out, on a lower level of abstraction, the client part of the Externalization Service implementation can be exploited by the Persistent Service to access the object's persistent state with no need to equip client objects with additional methods used by PDS protocol.

Acknowledgements

The authors of this paper would like to express their thanks to Jaromír Adamec and Christian Bac for many valuable comments and suggestions. Also, Antonín Brčák, Michal Fadljevič, Michael Gróf, and Nguyen Duy Hoa deserve a special credit for taking part in the implementation.

References

[AJJ+ 92] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski: Transparent object migration in COOL-2. Technical report, Chorus Systemes and Universite Pierre et Marie Curie, 1992.

[AJL92] P. Amaral, C. Jacquemot, and R. Lea: A model for persistent shared memory addressing in distributed systems. Technical report, Chorus systemes, 1992.

[AK85] D. A. Abramson and J. K. Keedy: Implementing a large virtual memory in a distributed computing system. In Proceedings of the 18th Hawaii Int. Conference on System Sciences, pages 515-522, 1985.

[ALJ92] P. Amaral, R. Lea, and C. Jacquemot: Implementing a modular object oriented operating system on top of CHORUS. Technical report, Chorus Systemes, 1992.

[ANS94] American National Standards Institute ANSI: Basic Reference Model of Open Distributed Processing-Part 3: A Perspective Model, 1994.

[BM92] A. L. Brown and R. Morrison. A Generic Persistent Object Store. Software Engineering Journal, 7(2) pp. 161-168, 1992.

[Bou94] F. Bourdon: The Automatic Positioning of Objects in COOL V2. Technical report, Service d'Etudes communes des Postes et Telecommunications, 1994.

[CBHS93] V. Cahill, S. Baker, C. Horn, and G. Starovic: The Amadeus GRT - Generic Runtime Support for Distributed Persistent Programming. Proceedings of OOPSLA-93, pages 144-161, 1993.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg: Distributed Systems Concepts and Design. Addison-Wesley, second edition, 1994.

[Cor95] Oracle Corporation: Object Query Service OMG 95-1-3. 1995.

[CP89] D. E. Comer and L. L. Peterson: Understanding Naming in Distributed Systems. Distributed Computing, 3(2), 1989.

[dA93] P. Amaral: PAS: A Framework for studying the implementation of multiple address spaces. PhD thesis, Universite Pierre et Marie Curie - Paris, 1993.

[DdBF+92] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan: Grasshopper: An orthogonally Persistent Operating System. Computer Systems, 7(3) pp. 289-312, 1992.

[DOM93] DOME User Guide, Object-Oriented Technologies Ltd., 1993.

[DRH+ 92] A. Dearle, J. Rosenbergr, F. Henkens, F. Vaughan, and K. Maciunas: An Examination of Operating System Support for Persistent Object System. In 25th Hawaii International Conference on System Services, volume 1, pages 779-789. IEEE Computer Society Press, 1992.

[Ede92] D. R. Edelson: Smart Pointers: They’re Smart but They’re Not Pointers. UCSC-CRL-92-97, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, 1992.

[Fra95] Framingham Corporate Center: Object Services RFP 5. OMG 95-3-25, 1995.

[FS94a] P. Fereira and M. Shapiro: Garbage Collection and DSM Consistency. Technical report, INRIA, 1994.

[FS94b] P. Ferreira and M. Shapiro: Garbage Collection of Persistent Objects in Distributed Shared Memory. Technical report, INRIA, 1994.

[HCF+ 95] D. Haginont, P. Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossiere, and X. R. de Pina: Persistent Shared Object Support in the Guide System - Evaluation and Related Work. Technical report, Bull-IMAG/Systems, 1995.

[HK93] G. Hamilton and P. Kougiouris: The Spring nucleus: A microkernel for objects. In Proceedings of the 1993 Summer Usenix Conference, June 1993.

[HKPCS95] J. Hans, A. Knaff, E. Perez-Cotes, and F. Saunier: Arias: Generic Support for Persistent Runtimes. Technical report, Bull-IMAG/Systems, 1995.

[HMA90] S. Habert, L. Mosseri, and V. Abrossimov: COOL: A Kernel Support for Object-Oriented Environments. Technical report, INRIA and Chorus Systemes, 1990.

[HRH95] M. Hollins, J. Rosenberg, and M. Hitchens: Breaking the Information Hiding Barrier. Technical report, University of Sydney and University of Western Sydney and Australia, 1995.

[IBM93a] IBM Corp. SOMobjects Developer Toolkit - Collection Classes Reference Manual, 1993.

[IBM93b] IBM Corp. SOMobjects Developer Toolkit - Programmers Reference Manual Version 2.0, 1993.

[KB92] J. K. Keedy and P. Broessler: Implementing Databases in the MONADS Virtual Memory. In Proceedings of the Fifth International Workshop on Persistent Object Systems, Springer Workshops in Computing series, San Miniato (Pisa) Italy, 1992.

[KN93a] Y. A. Khalidi and M. N. Nelson: A Flexible External Paging Interface. In Proceedings of the Usenix conference on microkernels and other architectures, September 1993.

[KN93b] Y. A. Khalidi and M. N. Nelson: The Spring Virtual Memory System. Technical Report SMLI-93-9, Sun Microsystems, 1993.

[KR89] J. L. Keedy and J. Rosenberger: Support for objects in the MONADS architecture. In Proceedings of the Int. Workshop on Persistent Systems, pages 202-213, 1989.

[LXC93] S. B. Lim, L. Xao, and R. Campbell: Distributed Access to Persistent Objects. Technical report, University of Illinois at Urbana Champain, Department of Computer Science, 1993.

[MA90] R. Morrison, M. P. Atkinson: Persistent Languages and Architectures. In Security and Persistence, J. Rosenberg and J. L. Keedy(ed.), Springer, pages 9-28, 1990

[Mey88] B. Meyer: Object-Oriented Software Abstraction, Prentice Hall, 1988

[MCCK94] R. Morrison, R. C. H. Connor, Q. J Cutts, and G. N. C. Kirby: Persistent Possibilities for Software Environments. IEEE Computer Society Press, pages 78-87, 1994.

[MGH+ 94] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia: An Overview of the Spring System. In Proceedings of Compcon Spring 1994. IEEE, February 1994.

[MGNS91] M. Makpangou, Y. Gourhant, J. P. Le Narzul, and M. Shapiro: Fragmented Objects for Distributed Abstraction. Technical report, INRIA, 1991.

[Mul94] S. Mullender, editor: Distributed Systems. Addison-Wesley, second edition, 1994.

[OMGa] Common Object Model Specification, Draft version 0.2, OMG 94-10-9, 1994.

[OMG90] Object Management Architecture Guide 1.0, OMG 90-9-1, 1990

[OMG91] Draft Common Object Request Broker Architecture Revision, OMG 91-1-1, 1991.

[OMG92] Object Service Architecture, OMG 92-8-4, 1992.

[OMG92a] Kala-Standardizing on Object Meta Services, Brief Response to the OMG services, Request for Information, OMG 92-4-5, 1992.

[OMG93] IDL C++ Language Mapping Proposal, OMG 93-4-4, 1993.

[OMG94a] Common Object Services Volume I, OMG 94-1-1, 1994.

[OMG94b] Persistent Object Service Specification, OMG 94-10-7, 1994.

[OMG94c] Comparsion of the OMG and ISO-FCCITT Object Models, OMG 94-12-30, 1994.

[OMG94d] Universal Networked Objects, ORB 2.0 RFP Submission, OMG 94-9-32, 1994.

[OMG94e] Relationship Service Specification, Joint Object Services Submission, OMG 94-5-5, 1994.

[OMG94f] Compound LifeCycle Addendum. Joint Object Services Submission. OMG 94-5-6, 1994.

[OMG94g] Object Externalization Service. OMG 94-9-15, 1995.

[OMG95a] ODP Trading Function Final Draft, ISO/IEC DIS 13235, OMG 95-7-6, 1995.

[OMG95b] Object Services RFP 5. OMG TC Document 95-3-25, 1995.

[OMK93] D. B. Orr, R. W. Mecklenburg, and R. Kuramkote: Strange Bedfellows: Issues in Object Naming Under Unix. In Proceedings IEEE93, pages 141-145, Department of Computer Science, University of Utah, Salt Lake City, UT 84112, USA, 1993.

[ORBIXa] Orbix, Programmer's Guide. IONA Technologies Ltd. Dublin, 1994

[ORBIXb] Orbix, Advanced Programmer's Guide. IONA Technologies Ltd. Dublin, 1994.

[PSWL94] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little: The Design and Implementation of Arjuna. Technical report, Computing Laboratory, University of Newcastle upon Tyne, 1994.

[RHB+ 90] J. Rossenberg, F. Henskens, A. L. Brown, R. Morrison, and D. Munro: Stability in a Persistent Store Based on a Large Virtual Memory. In Security and Persistence, Workshops in Computing, pages 229-245. Springer-Verlag, 1990.

[SDP92] M. Shapiro, P. Dockman, and D. Plainfosse: SSP Chains: A Robust Distributed References Supporting Acyclic Garbage Collection. Technical report, INRIA, 1992.

[SDP93] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington: An Overview of the Arjuna Distributed Programming System. Technical report, Computing Laboratory, University of Newcastle upon Tyne, 1993.

[SF94] M. Shapiro and P. Ferreira: Larchant-RDOSS: A distributed shared persistent memory and its garbage collector. Technical report, INRIA, 1994.

[SG91a] S. S. Simmel and I. Godard: The Kala Basket - A Semantic Primitive Unifying Object Transactions, Access Control, Versions and Configurations. In OOPSLA, 1991.

[SG91b] S. S. Simmel and I. Godard: The Kala Basket: A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations. In OOPLSA'91, pp. 230-246, 1991.

[SGH+ 91] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot: SOS: An Object-Oriented Operating System - Assessment and Perspectives. Technical report, INRIA, 1991.

[SGM89] M. Shapiro, P. Gautron, and L. Mosseri: Persistence and Migration for C++ Objects. In ECOOP'89, Nottingham (GB), July 1989.

[Sha94a] M. Shapiro: A Binding Protocol for Distributed Shared Objects. Technical report, INRIA, 1994.

[Sha94b] M. Shapiro: Structure and Encapsulation in Distributed Systems: The Proxy Principle. Technical report, INRIA, 1994.

[Sim92] S. S. Simmel: Providing commonality while supporting diversity. Hotline on Object-Oriented Technology, 3(10), August 1992.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson: Texas: An Efficient, Portable Persistent Store. In A. Albano and R. Morrison, editors, Persistent Object Systems, Workshop In Computing, pages 11-33. Springer-Verlag, 1992.

[SPFA94] M. Shapiro, D. Plainfosse, P. Ferreira, and L. Amsaleg: Some Key Issues in the Design of Distributed Garbage Collection and References. Technical report, INRIA, 1994.

[Str94] D. Stroustrup: The C++ Programming Language. Second Edition. Addison-Wesley, 1995.

[SUZ95] M. Steinder, A. Uszok, K. Zielinski: A Framework for Inter-ORB Request Level Bridge Construction. Submitted to the ICDP'96 - IFIP/IEEE International Conference on Distributed Platforms, Dresden, Germany, 1996.

[Tan95] A. S. Tanenbaum: Distributed Operating Systems. Prentice Hall, 1995.

[VD92] F. Vaugham and A. Dearle: Supporting Large Persistent Stores using Conventional Hardware. In A. Albano and R. Morrison, editors, Persistent Object Systems, Persistent Object Systems, Workshop In Computing, pages 35-53. Springer-Verlag, 1992.