Lessons Learned from Implementing the CORBA Persistent Object 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) 2191 4266
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
180 00 Prague 8
Czech Republic
phone: (42 2) 6605 3291
fax: (42 2) 858 5789
e-mail: {kleindie, plasil}@uivt.cas.cz

Abstract. In this paper, the authors share their experiences gathered during the design and implementation of the CORBA Persistent Object Service. There are two problems related to a design and implementation of the Persistence Service: first, OMG intentionally leaves the functionality core of the Persistence Service unspecified; second, OMG encourages reuse of other Object Services without being specific enough in this respect. The paper identifies the key design issues implied both by the intentional lack of OMG specification and the limits of the implementation environment characteristics. At the same time, the paper discusses the benefits and drawbacks of reusing other Object Services, particularly the Relationship and Externalization Services, to support the Persistence Service. Surprisingly, the key lesson learned is that a direct reuse of these Object Services is impossible.

1 Introduction

1.1 CORBA and TOCOOS

Around 1990, the Object Management Group (OMG) introduced the Objects Management Architecture (OMA) for distributed systems [OMG95c], which defines an *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 Orbix [ORBIXa, ORBIXb], SOM [IBM94a, IBM94b], DOME [DOM93], NEO [NEO96], and HP ORB+ [HP95],

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 a *target* (or *server object*) provided by a server. The functionality of server objects is specified via the Interface Description Language (IDL) defined in [OMG95]. In a client, the IDL compiler typically allows a server object to be used as an ordinary object (by generating an access to a stub or a proxy in the client); the abstraction provided for accessing a server object via a stub (proxy), both specified by the same IDL interface, is referred to as *CORBA object* [OMG94a]. Generally, the request format is vendor-dependent. The CORBA 2.0 standard [OMG94d] specifies ways to interconnect different CORBAs either by

[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

transforming requests in a bridge/gateway or by transporting requests in the standardized Internet Inter-ORB Protocol (IIOP).

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 (Object Lifecycle Service), looking up a server with a specific interface in another CORBA environment (Object Trading Service), or managing persistent objects (Persistent Object Service). The functionality of an Object Service is specified as a set of interfaces specified in IDL, e.g. [OMG94a, OMG94b]. There are two design principles that OMG follows: first, OMG intentionally leaves services not fully specified; second, services may be mutually dependent and, at the same time they should be able to exist separately, thus partially covering functionality of other services. Typically, an Object Service is used by inheriting a subset of IDL interfaces specifying the Object Service.

Since 1994, we have participated 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 the Object Services that would furnish the bridge with enhanced functionality, such as persistence 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

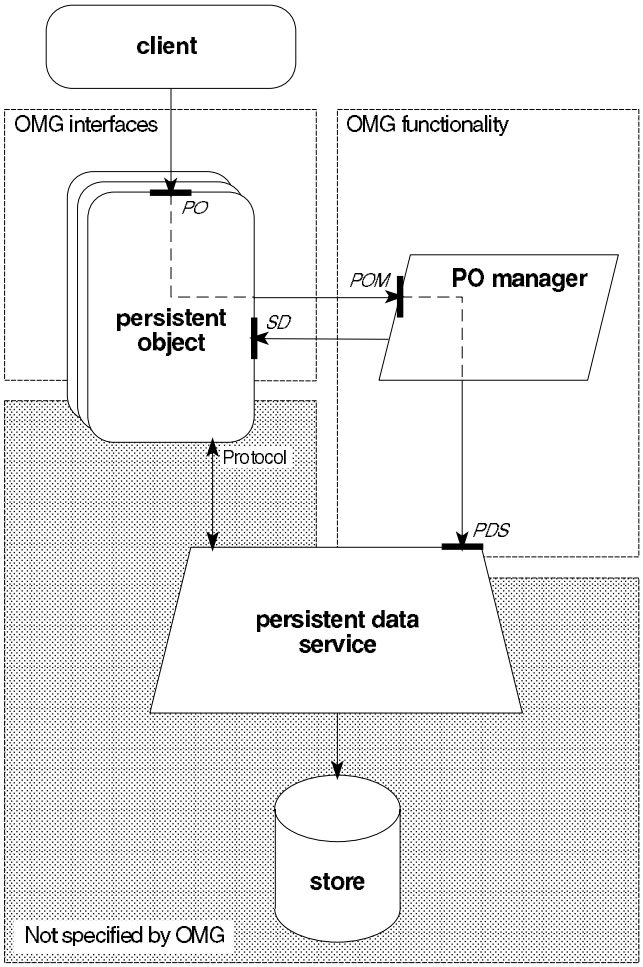

Figure 1 Standardized and unresolved parts of the OMG Persistence Service

the CORBA Persistent Object Service (the Persistence Service for short; similarly, we will skip "object" in the names of other CORBA Object Services). 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 corresponding OMG document [OMG94b] leaves the design of the Persistence Service functionality core unspecified, the first goal of the paper is to analyze those important issues that have been left unresolved by the OMG specification and to report on lessons we have learned from our design and implementation. Our second goal is to share the lessons we have learned when trying to follow strictly the recommended OMG architecture strategy which strongly encourages reusing other Object Services to minimize duplication in functionality. In compliance with this recommendation, in our design and implementation, we focused on the Relationship Service [OMG94e] and the Externalization Service [OMG94g]. As the inter-dependencies among the three Object Services are rather complex and potentially circular, we will also aim at providing the reader with an analysis of inherent trade-offs.

1.3 Structure of the paper

The paper has the following structure: Section 2 is focused on our project requirements and restriction. It also very briefly summarizes the OMG Persistence Service specification. In Section 3, we focus on our Persistence Service design decision. We provide the reader with the decision we made with respect to the general issues associated with the design and implementation of the Persistence Service - determining object persistence, accessing object attributes, updating object state, resolving dependencies and referential integrity, etc. Also, our approach to the key design architecture issues is addressed. Our implementation of the Persistence Service is described in Section 5 with a particular emphasis 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 Persistence 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 design and implementation of the Persistence Service, especially while balancing necessary trade-offs, and particularly while analyzing the option of reusing other Object Services in our implementation of the Persistence Service.

2 Project requirements and restrictions

2.1 Our design goals and boundaries

From the very beginning of our Persistence Service design, we had to consider two essential properties of the target application, the bridge. First, as the bridge uses CORBA distributed objects as well as local C++ objects, the Persistence Service implementation must be able to handle both kinds of objects. (We have, therefore, found it natural to consider implementing the Persistence Service only for the C++ environment.) Second, parts of the bridge must run in different CORBA environments. The Persistence Service implementation must not depend on any CORBA implementation-specific properties of the target environment.

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

a) The implementation should fully comply with the OMG Persistence Service specification [OMG94b].
b) The implementation should not depend on any other service unless a suitable implementation of it is readily available.
c) The implementation should neither require any change to the C++ [Str94] and IDL [OMG95] languages, nor any modification of the hosting CORBA environment.

2.2 The OMG specification of the Persistence Service

The Persistence Service is specified in [OMG94b], where 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()*.

A PDS supports a collection of pairs <*Datastore*, *Protocol*>. *Datastore* actually saves and loads the PO's data, *Protocol* describes the way a 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, a 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.

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

a) a POM knows about all available PDSs and the combinations of *Datastore* and *Protocol* that 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, the POM knows which *Protocol* the PO supports (implementation: the supported *Protocol* is deduced for example from the PO's type),

the POM resolves the PDS in the following steps:

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

2.3 Reusing other services

One of the CORBA architectural strategies is to stimulate mutual reuse among Object Services. More specifically, [OMG95b] reads: *"Each service should build on previous services when appropriate"* with the key motivation stated as *"Functionality should belong to the most appropriate service."* With respect to the Persistence Service, the corresponding OMG specification discusses the option of integrating the Persistence Service with other Object Services. The discussion distinguishes the services that potentially may use the Persistence Service, such as the Backup/Restore Service or the Replication Service, and the services that may be used by the Persistence Service, such as the Externalization Service or the Relationship Service. At first glance, it appears that reusing the latter two services might cover a substantial portion of a Persistence Service implementation. Therefore, we originally followed this track recommended by OMG. We summarize the lessons we learned during this stage of the design in Section 5, where we discuss the pros and cons of integrating these two services into our implementation of the Persistence Service and also provide the reasons why we have not done it in the current implementation.

3 Persistence Service design decisions

Filling the semantic gap left in the OMG specification of the Persistence Service (Section 2.2) involves many decisions to be taken when designing and implementing the service. In this section, as a result of both thorough analysis made before our decisions were made and experience gained from the actual design and implementation, we provide an overview of those design issues and tradeoffs we feel that everyone who implements the Persistence Service will have to face. We divide these issues into two parts: *general design issues* (determining an object's persistence property, accessing persistent object state, updating persistent object state, resolving dependencies) explained in Section 3.1, and *design architecture issues* (basically the internal structure of the building blocks from Figure 1) analyzed in Section 3.2.

3.1 General design issues

3.1.1 Key concepts

As for *persistence* [e.g. Tan95, Mul94, CDK94, SKW92, DdBF+92] 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: A *persistent object* is an object, the lifetime of which can exceed the lifetime of the application it is used in. *Persistent state* (of a persistent object) is the n-tuple of values corresponding to the n attributes of the object. 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 [Mey88, Str94].

3.1.2 Determining object persistence property

Basically, there are three ways to determine objects' persistence properties: *static*, *semidynamic* and *dynamic determination*.

Static determination: At compilation time, certain application objects may be statically denoted as being persistent, typically by inheriting from a persistent base class. Such objects will retain their persistence 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 persistence property statically and the impossibility of deactivating it later is not very convenient for the user. Thus, this black-or-white approach is not desirable.

Semidynamic determination: This approach is a modification of static determination through 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 objects' persistence. This control is limited to those classes of objects which have been statically preselected [Mey88], e.g. by inheriting from a "PersistentObject" base class.

Dynamic determination: This highly desirable approach allows the user to decide dynamically the persistence property of all objects (*orthogonal persistence* [MA90]).

To be able to implement the dynamic determination approach that guarantees persistence for all objects, the C++ compiler used would have to provide ways for accessing runtime type information for each C++ class. Since this is usually not the case with the C++ compilers and since we obliged ourself not to modify any C++ compiler, we decided to choose the semidynamic approach, e.g. support persistence only for certain, statically preselected, objects. Such objects are derived from a common base class and must explicitly export the runtime type information required by the Persistence Service. This information includes a list of the object's non-reference attributes, a list of the object's references pointing to other objects, and the identifier of the object's class.

3.1.3 Accessing object attributes

When manipulating an object's persistent state, two principal approaches to accessing its attributes can be identified: *access via object's methods* and *direct access.*

Access via object's methods: Typically, each persistent object is equipped with two specialized methods for saving and loading its state. Basically, the methods can be made public, and thus a part of the object IDL interface, or they can be made accessible to the underlying persistence system only - e.g. by using the friend construct in C++. While making them public works nicely also for distributed calls, the former alternative is limited for the server side only.

Direct access: Access to the object's attributes is done by direct access to the memory location at which the object is stored. This approach is typically used when a persistent address space is supported. A number of approaches towards the persistent address space architecture exist. The most well-known of these are [AJJ+92, AJL92, PSWL94, SDP93, SF94, FS94b, SG91a, SKW92, DdBF+92, and HCF+95].

In this case, there was basically no problem with the decision since the limitations of a distributed environment set heavy odds against the direct access approach. Thus, every persistent object in our implementation is equipped with two methods, *save_state()* and *load_state()*, defining how to store and load the object's attributes compactly.

3.1.4 External representation data format

The format of data external representation can be either *canonical* or *ad hoc*.

Canonical format: Data on the external media is stored in a standardized format, possibly the *Standard Stream Data Format* defined in [OMG94g] with the bound IDL type set and identification keys reused from the Naming Service. Such a format, of course, has the advantage of being widely recognized and can be used for porting objects' data from one implementation to another. On the other hand, supporting such a format sometimes requires sacrifices in the implementation and may slow down

overall performance.

Ad hoc format: Each datastore implementation comes with its own format for externalized data, typically tailored for a particular save/load protocol and naturally taking advantage of the underlying media representation (files, raw blocks, database tables).

In our design we came up with the ad hoc format that suits well our datastore implementation and supports the save/load protocol we use.

3.1.5 Updating persistent object state

The updating of an object's persistent state can be either *automatic* or *explicit*.

Automatic: Updating is system-controlled, based on a consistency protocol which can be e.g. transaction based, virtual memory based, system event based, etc. [AJL92, HKPCS95].

Explicit: Updating is application-controlled, i.e. *store* and *restore* methods of the Persistence Service are called explicitly by the application.

We chose the explicit approach for the current implementation. Future extension to provide the automatic updating is possible, probably transaction based and implemented via Orbix filters.

3.1.6 Resolving dependencies and referential integrity

In addition to the issue of when to update the persistent state of an object, the question arises of how the object's dependencies are handled: the object's dependencies are either not considered (*shallow updating*), or they are considered. In the latter case, the dependencies can be updated recursively up to the n-th level; in general, to ensure referential integrity, the transitive closure of dependencies is considered (*deep updating*). To implement deep updating, object dependencies have to be resolved; two techniques are used for this purpose: *grouping* and *evaluation of dependencies.*

Grouping: The current status of relations among objects is not evaluated; dependencies are resolved implicitly by defining groups of object. Such a group must be closed with respect to inter-object references. As examples, see *clusters* in COOL [HMA90, AJJ+92, ALJ92] or *groups* in SOM [IBM94a, IBM94b].

Evaluation of dependencies: Dependencies are explicitly evaluated (typically, before each update of an object's persistent state). This can be either application-controlled or system-controlled (based on hardware-supported reference identification). In the latter case, the techniques of *early* or *late pointer swizzling* are typically used [SKW92, VD92].

Having possible integration with the Relationship Service in mind, we selected the evaluation of dependencies technique for our implementation, since it emulates the graph traversal mechanism used by the Traversal Criteria object from the Relationship Service.

3.2 Design architecture issues

This section briefly describes three major components of our Persistence Service proposed architecture: Persistent Data Service, Protocol, and Data Store. In accordance with [OMG94b], the Persistent Data Service is responsible for actually carrying out all Persistence Service requests. As discussed in Section 3.1.3, the Persistent Data Service accesses the persistent attributes of an object using the *POProtocol* interface (this interface is a part of protocol in terms of [OMG94b]). The object dependencies are processed together with an object (Section 3.1.6). The heart of the Persistent Data Service architecture is, therefore, an algorithm capable of recursively traversing all nodes of a persistent object graph, together with a list of all persistent objects being served by the Persistent Data Service.

The Protocol describes a method of communication between the Persistent Data Service and persistent objects. It is designed to provide the Persistent Data Service with the necessary information, while remaining as simple as possible from the persistent object's point of view. As our environment does not provide sufficient information about an object's type, we have included a method to get a unique type ID as a part of the Protocol.

The only responsibility of the Data Store is persistent storage management, together with reading and writing the managed data. As the structure of the persistent storage may vary depending on the application environment, the Data Store interface should be flexible enough to cover a wide range of underlying storage mechanisms. We have, therefore, decided to keep the Data Store interface very simple, providing only basic methods to read and write untyped data.

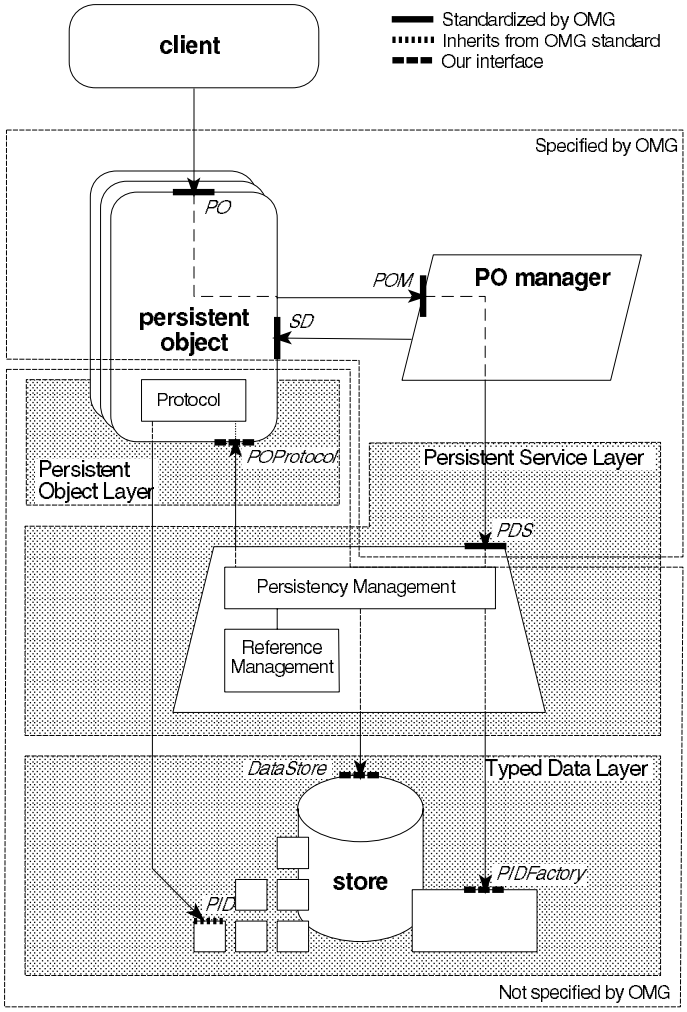

Figure 2 Proposed architecture of the OMG Persistent Service

4 Persistence: our design approach

4.1 Our design architecture

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

Figure 2 depicts the structure of our implementation in terms of the OMG Persistence Service specification. As indicated in this figure, the three layers of our architecture roughly correspond to the major components of the OMG specification: the Typed Data Layer implements the OMG Datastore component, the Persistent Object Layer contains important sections of the OMG Protocol, and the Persistence Service Layer represents both the OMG Persistent Data Service and the OMG Persistent Object Manager components.

4.2 Typed Data Layer

This layer is embodied by the Store Access Module responsible for accessing data on external storage media. The module implements storage access primitives and is also capable of providing single

level transactions necessary for crash recovery support in higher layers. The module consists of four interfaces. The first two interfaces, called *PID* and *PIDFactory*, introduce the notion of a unique persistent identifier. The other two, called *DataStore* and *DataStoreFactory*, provide a means of accessing the external store:

```
module StoreAccess {
    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. 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 {
    void clear (in ulong location);
    oneway void remove ( );
    boolean is_empty (in ulong location);
    void storeAt
        (in tBuffer buf, ulong location, in ulong offset);
    void restoreAt
        (inout tBuffer buf, ulong location, in ulong offset);
};
```

The *storeAt()* method is used to put data into a store location denoted by the *PID* itself and by the *location* and *offset* arguments, the *restoreAt()* method is used to retrieve data in a similar manner. A location can be emptied using the *clear()* call, or tested whether it is empty using the *is_empty()* call. 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):

```
interface PIDFactory {
    PID get_root_PID ( );
    PID create_unique_PID ( );
    PID create_PID_from_string (in string pid_string);
};
```

As PIDs are devised by the store itself, 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 purpose of maintaining a directory of saved data. It is up to the client to specify a strategy for root record usage.

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

```
interface DataStore {
    void trans_begin ( );
    void trans_commit ( );
    void trans_abort ( );
    PIDFactory get_PIDFactory ( );
    void destroy_on_remove ( );
};
```

As the Persistence 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 the means to obtain an appropriate *PIDFactory* by calling the *PIDFactory::get_PIDFactory()* method. The *DataStore* instances used by the Persistence service are manufactured by an instance of the *DataStoreFactory*:

```
interface DataStoreFactory {
    DataStore open_DataStore (in string media_ID);
};
```

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.

4.3 Persistent Object Layer

4.3.1 Object Storage Module

This module is responsible for saving and loading the persistent attributes of an individual object and for exporting type and object reference information. As the only entity authorized to access a persistent object's internal state is the object itself, the services of this module are exported in the 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 module contains one interface:

```
module ObjectStorage {
    interface POProtocol { };
};
```

In order to be able to cooperate with the Persistence 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);
    void 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 runtime type information to the user, the *POProtocol* interface exports the *get_TID()* method to do this - this method returns a user-supplied identifier representing 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()* and *set_references()* methods load and save the references to other persistent objects. When creating the list of references, the method *get_references()* is expected to call the *get_references()* methods of its direct superclasses (transitively); the list *references* is thus created from the contributions implied by the class hierarchy structure. Similarly, the dual method *set_references()* calls the methods *set_references()* in the direct superclasses (transitively). Each such call is supposed to eliminate the references it has processed from the list *references* passed to it as the actual inout parameter.

4.3.2 Object Factory Module

When loading dependencies of a persistent object (Section 3.1.1), 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 to be capable of creating an object instance of the type corresponding to a given TID. In practice, the generic factory interface is more convenient for the purpose of obtaining specific object instances if it is extended by a mechanism for registering specific object factories.

4.4 Persistence Service Layer

This layer 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.

4.4.1 Reference Management Module

As the implementation of the Persistence Service is expected to work with C++ objects as well as with CORBA objects, we can not use CORBA object IDs to identify object instances. We need another kind of identifier associated with every persistent object to encode inter-object references in the persistent store. Fortunately, there is no need to devise another object identification mechanism. 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 a saved location identifier. In an application, we need to prevent the Persistence Service from creating several object instances using the same persistent object image in the datastore. As a 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 already been loaded.

4.4.2 Persistence Management Module

This module is the heart of the Persistence Service functionality. It exports the following specialized version of the Persistent Data Service *CosPersistencePDS::PDS* interface:

```
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()* and *disconnect()* method turn on and 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 3.1.5).

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 *store()* and *restore()* methods process objects' dependencies recursively. When saving an object, the Persistent Data Service starts by finding a PID associated with the object. A new PID is assigned if necessary. The object's TID 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 a *delete()* call is issued, an 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.

4.5 Design evaluation

We made many key design decisions based on limited environment characteristics (Section 2.1). Thus, anyone facing the task of designing a Persistent Object Service in a similar setting will probably ask similar questions as we did, and may therefore benefit from our design decisions (Section 3) and the proposed architecture (Section 4).

In a different environment, the general architecture and some design details may still remain useful, while permitting the developer to enhance certain environment-specific features. As an example, we have considered a system with runtime type information support - in such a system, the *POProtocol* interface can be modified to exploit the type information, thus relieving the user of the necessity to specify the information by hand. Another example could be a system with virtual memory support, in which memory management can be used to implement delayed loading of object dependencies, removing the necessity to process object graphs in one piece.

At this point, let us emphasize the key points of our solution with respect to its performance impact. The ability of the *PID::storeAt()* method to place data at the specified location in the Data Store extends the power of the data manipulation semantics by increasing the level of the data handling granularity, thus improving performance in the cases when only a small subset of an object's attributes needs to be saved. The persistent data is saved in an ad hoc format, taking advantage of the Data Store architecture and thus obviously achieving better average storing/restoring time compared to the Standard Stream Data Format (Section 3.1.4). Also, explicit providing of the *POProtocol::save_state()* and *POProtocol::load_state()* methods gives the user the possibility to save/load only the attributes that he/she considers necessary. On the contrary, if the direct access approach (Section 3.1.3) were employed, the Persistence Service would have to save/load also non-persistent (limited lifetime, Section 3.1.1) attributes, such as large transient buffers and file handles. On the other hand, the necessity to load *all* objects from the Data Store to memory at one shot reduces performance in the case that only a small subgraph of the whole transitive closure of dependencies is required by the application.

5 Reuse of other Object Services in the Persistence Service

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

5.1 Reusing Relationship Service

5.1.1 Building Persistence Service over Relationship Service

The goal of the Relationship Service [OMG94e] is to provide tools for operating upon abstractions based on entity relationship diagram concepts. According to the abstractions it provides, the Relationship Service is hierarchically structured into 3 levels. The base level relationship provides *Role* and *Relationship* as a means for organizing entity objects (*related objects*) in entity-relationship-diagram-like structures. The second level provides *Node*, *Graph*, and *Edge* which can be used to create graphs of related objects. It also provides *Traversal* and *TraversalCriteria* for traversing these graphs and, very importantly, defining subgraphs at runtime. A *TraversalCriteria* object can use the concept of *propagation value*. Finally, the highest level provides the specific relationships *Containment* and *Reference*. For brevity, we have found it useful to call an instance of the *Role*, *Relationship*, or *Node* interfaces an *r-object*.

When the Relationship Service is employed, two types of inter-object references are to be considered: inter-object references expressed by (i) standard object references and (ii) via r-objects. The concept of object dependencies (Section 3.1.1) covers 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 4.4.2, i.e. the transitive closure of dependencies is built by recursive evaluation of

//Setting pointer to HisObject

```
Object *MyObject,*HisObject;

MyObject->TheOther = HisObject;
```

//Traversing pointer

```
HisObject = MyObject->pointer;
```

//Creating relationship with HisObject

```
Node *MyNode;
Object *HisObject;
Role *MyRole,*HisRole;
Relationship *MyRelationship;
NamedRole NamedRoles (2);
RoleFactory *MyRoleFactory;
RelationshipFactory *MyRelFactory;

MyRole =
    MyRoleFactory->create_role (MyObject,IT_X);
HisRole =
    MyRoleFactory->create_role (HisObject,IT_X);
NamedRoles[0].name = strdup ("A");
NamedRoles[0].aRole = MyRole;
NamedRoles[1].name = strdup ("B");
NamedRoles[1].aRole = HisRole;
MyRelationship =
    MyRelFactory->create (NamedRoles,IT_X);
//IT_x serves to signal exception
```

//Traversing relationship

```
RelationshipHandle *MyRelHandle;

MyRelHandle.the_relationship = MyRelationship;
MyRelHandle.constant_random_id =
    MyRelationship->constant_random_id ( );
HisObject = MyRole->
    get_other_related_object (MyRelHandle,"B");
```

Figure 3 Increase in source code complexity

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 runtime, the traversal process should use the user-defined *TraversalCriteria* object (Section 5.1.2). In order to meet the minimum requirements of the Persistence 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 the general case, the two reference types can coexist in the transitive closure of an object's dependencies. This even allows several subgraphs defined by different *TraversalCriteria* objects to participate in one transitive closure. The Persistence Service should respect this flexibility in defining subgraphs. A way of specifying the *TraversalCriteria* in the Persistence Service interface must therefore be devised. Moreover, a mechanism has to be defined for updating the graph's persistent state by combining the effect of several store operations upon subgraphs of the graph. A dual problem arises with combining the effect of several restore operations upon subgraphs of the graph. A detailed study of these issues is a subject of our current research.

In theory, following strictly the Bauhaus principle 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 the transitive closure of dependencies, the overhead inherent in dereferencing only via r-objects would hardly be acceptable (Section 5.1.4). In addition, the Persistence Service needs to handle references among r-objects; to avoid infinite recursion, these references cannot be expressed by using r-objects. Therefore, the Persistence Service needs to handle standard object references in any case.

5.1.2 Making r-objects persistent

In this section, we will presume that both types of inter-object references coexist in the single transitive closure of an object's dependencies. How the standard object references are to be handled was described in Sections 4.3 and 4.4. To make the transitive closure persistent, however, r-objects themselves have to be saved/loaded as well. Thus, the key focus of this section is how to save/load r-objects.

As, in principle, the set of an object's attributes belonging to its persistent state is to be determined 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 by using *TraversalCriteria*. Consequently, to save r-objects, a special technique is necessary. In general, given a *TraversalCriteria* object and a root *Node*, the entire subgraph must be traversed to determine its relevant nodes and edges. Thus, saving of the subgraph can take place only after all the r-objects representing the subgraph are found. In the following we discuss the possible approaches to storing the r-object. The discussion is based on the sound assumption deduced from the OMG Relationship Service specification [OMG94e] that all *Roles* belonging to a given relationship maintain a reference to the particular *Relationship* object and - vice versa - the *Relationship* object keeps references to all its *Roles*. As an aside, such a reference can be a CORBA-reference. These references will be saved in the Data Store as PIDs. There are basically three ways of storing those r-objects:

a) **References in both directions are saved inside the persistent state of r-objects**: Both role -> relationship and relationship -> role references are saved as PID's inside the persistent state of the *Role* and the *Relationship* objects. This technique closely follows the principle of saving inter-object references as PIDs.

b) **References are stored outside the persistent state of r-objects**: Information about references among *Role* and *Relationship* objects is stored in a separate external data structure. 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*).

c) **A mixture of approaches a) and b) is used**. For example, relationship -> role references can be stored inside the persistent state of *Relationship* objects, and a list of the *Relationship* objects belonging to the graph is stored in a separate external data structure. At load time, role -> relationship references in roles are reconstructed via *Role::link()* calls.

The drawback of alternatives b) and c) is that they need a special mechanism to save references among r-objects. In our view, alternative a) is the most convenient one. It scales well, and it treats r-objects

in a similar way as ordinary objects. Naturally, in any case, each r-object has to support storing/restoring of its persistent state. Thus, with respect to reusing, the Relationship Service must be modified accordingly.

5.1.3 Advantages of reusing the Relationship Service

Naturally, the inherent benefit of building the Persistence Service upon the Relationship Service is reusing existing CORBA code. However, the dominant advantage of employing the Relationship Service in the Persistence Service is the power and flexibility of graph traversing operations. In the Relationship Service, it is very easy to make changes to searches for dependencies or groups of objects simply by changing the *TraversalCriteria* object. For that purpose, a set of specialized *TraversalCriteria* objects (e.g. those traversing only certain edge types) may be predefined. Furthermore, each *TraversalCriteria* object can be parametrized dynamically at runtime.

To benefit from this type of flexibility, the Persistence Service must provide its client with a way to specify a *TraversalCriteria* object to be used when traversing the persistent object graph. This can be achieved by enriching either the *POProtocol* or the *PDS* interface, the first option being more in line with the CORBA Persistence Service concept. In principle, the Persistence Service employing *TraversalCriteria* stores/restores a subgraph of a given graph. This raises the issue of what the semantics of combining subgraphs when storing/restoring parts of the given graph should be (subject to our current research).

5.1.4 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 *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 the Relationship Service may not be critical for the Persistence Service itself, as most of the time is spent operating with the Data Store. 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. The increase in source code complexity is obvious from the example in Figure 3.

According to our measurements made on a Sun

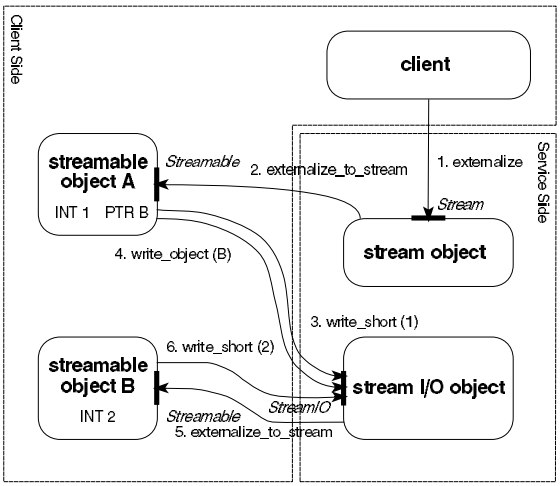

Figure 4 Simple Externalization Control Flow

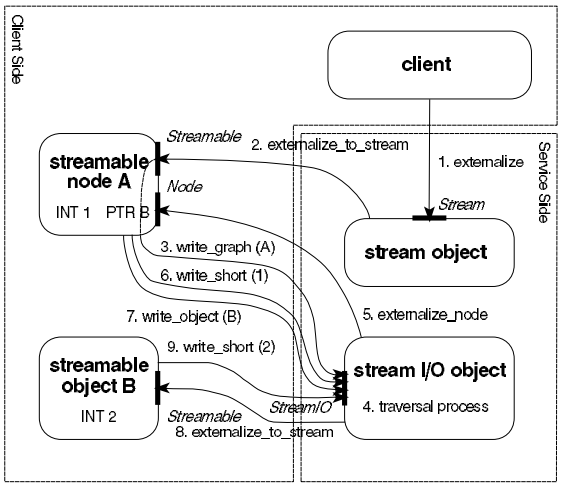

Figure 5 Compound Externalization Control Flow

SPARCstation 4 with Solaris 5.4 and Orbix 2.0, a call of a remote object's method with empty body with the target object being dereferenced via *Role::get_other_related_object()* is two to three times slower than calling the remote method directly. The results are even more convincing with the caller and the target object being in the same address space. Using *get_other_related_object()* of a server in the same address space slows the program down approximately 1000 times; using a server in a separate address space yields 10000 times slower execution. The Relationship performance can be substantially improved by reference caching, but even then there is a significant difference between the two cases.

5.2 Reusing Externalization

5.2.1 Relevant concepts of the Externalization Service

The Externalization Service supports the sequential saving/loading of objects from/into the CORBA environment. Externalized objects are saved on media in the canonical form described by [OMG94g]. Thus the Externalization Service allows an 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 from *Streamable* to make itself externalizable:

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

The methods take as a parameter a reference to a *StreamIO* object. This mechanism provides for dynamic binding between the object and the stream which externalizes the object'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 two methods for saving dependencies referenced from the object: *write_object()* and *write_graph()*. The *write_object()* method is used when the relations among objects are not represented via the Relationship Service. The typical control flow is illustrated in Figure 4. The *write_graph()* method is called when the Relationship Service is used to represent relations among objects (Figure 5). An Externalization Service capable of cooperating with the Relationship Service is called the *Compound Externalization Service* [OMG94g]. To distinguish the two cases, the non-compound Externalization Service will be referred to as the *Simple Externalization Service*. All in all, it should be emphasized that the *Stream* interface inherently implies a sequential way of saving and loading objects to and from external media.

5.2.2 Simple Externalization as a POS protocol

As stated in the Externalization Service specification [OMG94g, Section 3.1], the Externalization Service has been designed to be able to integrate with the Persistence Service as a specific POS protocol. At the same time, the reference to the Externalization Service in the Persistence Service specification [OMG94b, Section 6.17] reads: *"... the Persistence Service could use this service as a POS protocol."* However, neither of the OMG documents goes any further in specifying how the Externalization Service might be reused in the Persistence Service.

In fact, being inherently sequential at a higher level of abstraction, the Externalization Service can support the Persistence Service, inherently based on random access to individual objects, only in a very special case - when a Persistence 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. The following might be a fragment of the specialized implementation:

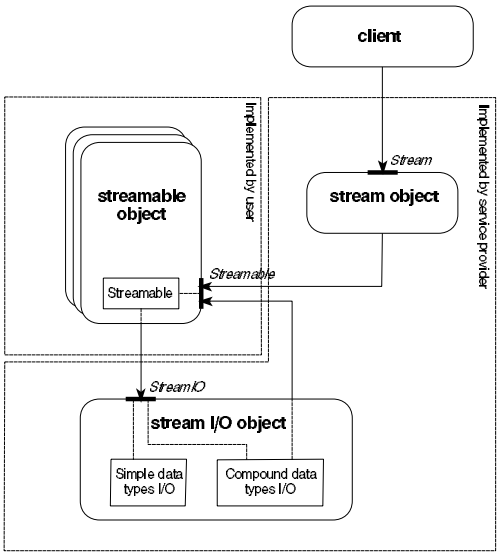

Figure 6 Externalization Service General Architecture

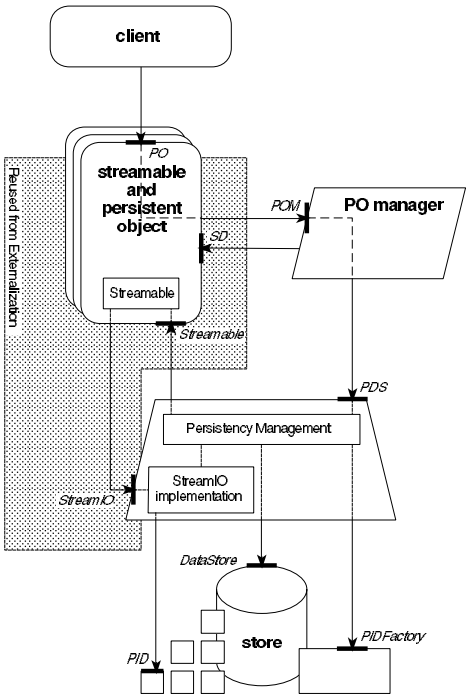

Figure 7 Reusing Externalization in Persistence Service

```
void StreamIO::write_char (char chr) {
   Buffer bfr;
   bfr->data = &chr;
   bfr->length = bfr->maximum = sizeof(char);
   CurrentPID->storeAt (bfr,0,CurrentOffset);
   bfr->data = NULL;
};

void StreamIO::write_object (Streamable obj)
{
   RelatedObjectsList->add (obj);
};
```

The `write_char()` method stores `chr` at the Data Store location defined by `CurrentPID`. Repeated calls of `write_object()` build a list of dependencies to be exploited after the *externalize_to_stream()* method returns (during the evaluation of the transitive closure of dependencies).

5.2.3 Advantages of reusing the Simple Externalization Service

Both the Persistence and the Externalization Services have a similar task: to store/restore the client object's attributes. This similarity can be exploited by using the *Streamable* interface in place of *POProtocol*, thus preventing a duplication of functionality. This is particularly advantageous when the client already uses the Externalization Service and is going to employ the Persistence Service. In such a case, the Persistence Service can use the existing *Streamable* implementation provided by the client's objects without any change; thus the Persistence Service does not need any additional protocol.

5.2.4 Disadvantages of reusing the Simple Externalization Service

Although exploiting the *Streamable* interface makes it possible to reuse parts of the client object's code, a large part of the Externalization Service remains unused. Obviously, the parts of the Externalization Service which reflect its sequential nature cannot be readily reused.

5.2.5 Reusing the Compound Externalization Service

So far, we have relied on calling the *Streamable* interface with an instance of a specialized *StreamIO* class specified as an argument; thus, the interface provided a direct access to an object's attributes without a need to change the *Streamable*

implementation. This trick, however, cannot be used with the Relationship Service's nodes, as the Compound Externalization Service uses a single *externalize_node()* call to externalize the node together with its roles. As all the roles adjacent to a node do not necessarily belong to the subgraph defined by the graph traversal process run by the Persistence Service, the roles not belonging to the subgraph do not have to be externalized. Thus, the standard implementation of the *CosCompoundExternalization::Node::externalize_node()* method prevents the Persistence Service from processing only the roles belonging to the subgraph. Providing a specialized implementation of the *externalize_node()* method could remedy this problem; this change, however, cannot be incorporated into the Externalization Service in such an elegant and flexible way as was the case with *StreamIO*.

6 Conclusion

The paper is based on our experience with designing and implementing the CORBA Persistence Service. As OMG leaves large parts of the Persistence Service functionality unspecified, several very important issues, such as handling of related objects and interfacing with a datastore, remain unresolved in the specification. Therefore, in Section 3, 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 also described in Section 3. The core functionality of our Persistence Service implementation is located in the Persistent Data Service layer, with the low level support routines split between the Persistence Object and Store Access layer. In Section 5 we provided the reader with relevant details and design evaluation.

Interesting lessons have been learned when evaluating the prospects of reusing the Relationship Service and the Externalization Service (Section 5). As for related objects, we have found it very important to treat both standard references and references defined by the Relationship Service in a unified way with respect to an object's dependencies. Therefore, in Section 3.1.1, 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 Persistence Service implementation alone; the user, however, may benefit from the ability of the Persistence Service to understand and process the graphs defined via r-objects of the Relationship Service. Thus, if the Relationship Service is implemented in a CORBA, the Persistence 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 Persistence 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 Persistence Service, which in contrast is inherently based on random access to individual objects, only in a very special case as discussed in Section 5.2. However, on a lower level of abstraction, the client part of the Externalization Service implementation can be exploited by the Persistence Service to access object's persistent state with no need to equip client objects with additional methods used by the PDS protocol. To reuse the Compound Externalization Service, an enhancement of its specification is necessary (Section 5.2.4).

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, Dušan Bálek, Antonín Brčák, Michal Fadljevič, Michael Gróf, and Nguyen Duy Hoa deserve special credit for taking part in the implementation. Finally, the authors are grateful to Adam Dingle and Camie Bates for proofreading the text.

References

[AJJ+92] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski: Transparent object migration in COOL-2. In Proceedings of Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems, ECOOP'92, Utrecht, June 1992

[AJL92] P. Amaral, C. Jacquemot, and R. Lea: A model for persistent shared memory addressing in distributed systems. Technical report TR-92-52, Chorus systemes, 1992.

[ALJ92] P. Amaral, R. Lea, and C. Jacquemot: Implementing a modular object oriented operating

system on top of CHORUS. In Proceeding s of OpenForum 92, Utrecht, November 92.

[A93] P. Amaral: PAS: A Framework for studying the implementation of multiple address spaces. PhD thesis, Universite Paris VI, 1993.

[Ben95] R. Ben-Nathar: CORBA: A guide to Common Object Request Broker Architecture. McGraw-Hill. 1995.

[Bou94] F. Bourdon: The Automatic Positioning of Objects in COOL V2. In Proceedings of 14th ICDCS, Poznan, IEEE Computer Society Press, June 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, ACM Press, 1993.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg: Distributed Systems. Concepts and Design. Addison-Wesley, 2nd Edition, 1994.

[CTP96] T. L. Casavant, P. Tvrdík, F. Plášil (Editors.): Parallel Computers: Theory and Practice. IEEE Press, 1996.

[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), pages 289-312, 1992.

[DOM93] DOME User Guide, Release 2.2, 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 Systems. In Proceedings of the 25th Hawaii International Conference on System Services, 1, IEEE Computer Society Press, 1992.

[Ede92] D. R. Edelson: Smart Pointers: They're Smart but They're Not Pointers. UCSC-CRL-92-27, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, 1992.

[FS94a] P. Ferreira and M. Shapiro: Garbage Collection and DSM Consistency. In Proceedings of the first symposium on the Operating Systems Design and Implementation Conference, Monterey, November 1994.

[FS94b] P. Ferreira and M. Shapiro: Garbage Collection of Persistent Objects in Distributed Shared Memory. In Proceedings of the Persistent Object Systems, Tarascon, September 1994.

[FS96] P. Ferreira and M. Shapiro: Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection, In Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, IEEE Computer Society Press, May 1996.

[HCF+95] D. Hagimont, 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. In Proceedings of the 9th Annual Conference on OO Programming Systems, Languages, and Applications, Portland, pages 129-144, October 1994.

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

[HKPCS95] J. Hans, A. Knaff, E. Perez-Cotes, and F. Saunier: Arias: Generic Support for Persistent Runtimes. In Proceedings of European Research Seminar on Advances in Distributed Systems, L'Alpe d'Huez, pages 220-226, April 1995.

[HP95] HP ORB Plus 2.0, URL: http//www.hp.com

[HMA90] S. Habert, L. Mosseri, and V. Abrossimov: COOL: A Kernel Support for Object-Oriented Environments. In Proceedings of the Joint ECOOP/OOPSLA Conference, Ottawa, pages 269-277, October 1990.

[IBM94a] IBM Corp. SOMobjects Developer Toolkit Users Guide, Version 2.1, 1994.

[IBM94b] IBM Corp. SOMobjects Developer Toolkit Programmers Reference Manual 2.1, 1994.

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

[KPT95] J. Kleindienst, F. Plášil, P. Tůma: Implementing CORBA Persistence Service, TR 117, Charles University Prague, Department of Software Engineering, 1995.

[KPT96] J. Kleindienst, F. Plášil, P. Tůma: CORBA and its Object Services. Invited Paper, SOFSEM'96, Springer LNCS (to appear), 1996.

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

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

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

[MoZa95] T.J. Mowbray, R. Zahavi: The Essential

CORBA, J. Wiley & Sons, 1995.

[MCCK94] R. Morrison, R. C. H. Connor, Q. J. Cutts, and G. N. C. Kirby: Persistent Possibilities for Software Environments. In Proceedings of the The Intersection between Databases and Software Engineering, IEEE Computer Society Press, pages 78-87, 1994.

[Mul94] S. Mullender, editor: Distributed Systems. Addison-Wesley, 2nd Edition, 1994.

[NEO96] Solaris NEO Operating Environment, Product Overview, Part No. 95392-003, Sunsoft Inc, March 96.

[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.

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

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

[OMG95] Common Object Request Broker Architecture and Specification Revision 2.0, OMG 96-3-4, 1995.

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

[OMG95c] Object Management Architecture Guide, 3rd Edition, R.M. Soley (Editor), John Wiley & Sons, 1990.

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

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

[PG95] F. Plášil, M. Gróf: An Overcoming of Inheritance Anomaly, TR 95-05-02, Department Informatique, Institut national des Telecommunications, Evry, France, 1995.

[PSWL94] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little: The Design and Implementation of Arjuna. BROADCAST Project deliverable report, 4, University of Newcastle upon Type, October 1994.

[PT95] F. Plášil, P. Tůma: Memory Management in Spring. TR 95-05-03, Department Informatique, Institut national des Telecommunications, Evry, France, 1995.

[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, 1990.

[SDP93] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington: An Overview of the Arjuna: A Programming System for Reliable Distributed Computing, IEEE Software, 8(1), pages 63-73, January 1991

[SF94] M. Shapiro and P. Ferreira: Larchant-RDOSS: A distributed shared persistent memory and its garbage collector. In Proceedings of the 9th Workshop on Distributed Algorithms (WADG), Le Mont Saint Michel, Sept. 1995.

[SG91a] S. S. Simmel and I. Godard: The Kala Basket: A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations. In Proceedings of OOPLSA'91, pages. 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. Computing Systems 2(4), 1989.

[Sha94a] M. Shapiro: A Binding Protocol for Distributed Shared Objects. In Proceedings of the International Conference on Distributed Computing Systems (ICDCS) , Poznan, June1994.

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

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson: Texas: An Efficient, Portable Persistent Store. In Proceedings of the Persistent Object Systems, San Miniato, A. Albano and R. Morrison (Editors), Springer, pages 11-33, 1992.

[Str94] D. Stroustrup: The C++ Programming Language. 2nd Edition. Addison-Wesley, 1995.

[SUZ96] M. Steinder, A. Uszok, K. Zielinski: A Framework for Inter-ORB Request Level Bridge Construction. In Proceedings of the IFIP/IEEE International Conference on Distributed Platforms, Chapman & Hall, Dresden, pages 86-99, 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 Proceedings of the Persistent Object Systems, San Miniato, A. Albano and R. Morrison (Editors), Springer, pages 35-53, 1992.