What We Are Missing in the CORBA Persistent Object Service Specification

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 the paper we try to summarize the weaknesses of the CORBA Persistent Object Service standard we felt were most significant while designing and implementing a Persistent Object Service compliant with the standard. The issues discussed in detail include: underspecified semantics of operations, underspecified functionality of POM, lack of "compound persistence", reusability of other services (relationship, externalization, compound externalization, and naming).

1 Introduction

At OOPSLA'96, we will present the paper "Lessons Learned from Implementing the CORBA Persistent Object Service" [KPT96b]. Compared to that paper, this contribution is more focused on the CORBA Persistent Object Service (POS) standard itself ([OMG94b]). It also tries to summarize the weaknesses of the standard we felt were most significant while designing and implementing a Persistent Object Service compliant with the Standard.

2 Overview of the POS Architecture

2.1 Goals

According to [Sess96], the Persistent Object Service (POS) specification (*the Standard* for short) was prepared as a trade-off based on the original IBM and SunSoft submissions to the OMG Request for Proposal (RFP) issued in 1992 ([OMG92b]). In addition to the requirements stated by the RFP, the POS specification was designed to meet the following goals [Sess96]: *support for corporate-centric datastores* (including databases of all types, filesystem based datastores, etc.), *datastore independence* (a single client API independent of a particular datastore; a single mechanism for storing/restoring objects to be used on the object server side), *open architecture* (new datastores to be plugged in at any time).

2.2 What the Standard says

As defined in the Standard, the POS architecture is based upon instances of the following components. *A Datastore* is an actual data repository with no predefined interface. The location of a persistent object in a datastore is determined by a *Persistent Identifier* (PID). A *Persistent Data Service* provides an uniform access to a datastore, and at the same time supports a mechanism called a *protocol* (also viewed as a POS component) for storing/restoring the persistent attributes of a persistent object. Another component, the *Persistent Object Manager* (POM), provides uniform access to different Persistent Data Service instances/types. The relationship of these components is summarized on Fig. 1 (basically taken from [OMG94b]). The goal of making the POS architecture open is achieved by allowing almost all meaningful *n:m* combination of these components (both instances and types). So an object and a PDS can support more protocols, a PDS can work with many datastores, more PDSs can have access to a particular datastore etc. (Fig. 2). Unfortunately, these options have to be read "between the lines" in the Standard.

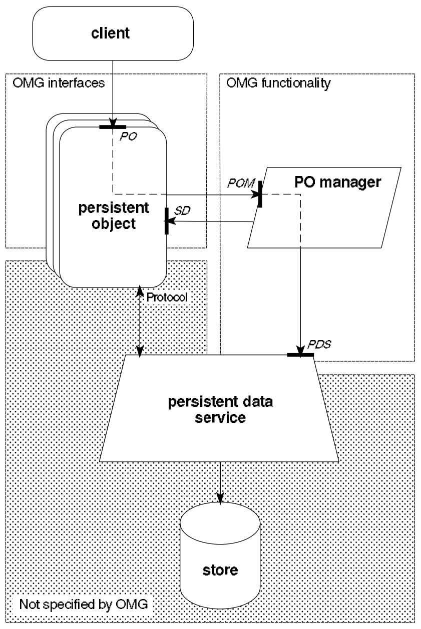

Figure 1 OMG Persistent Object Service structure

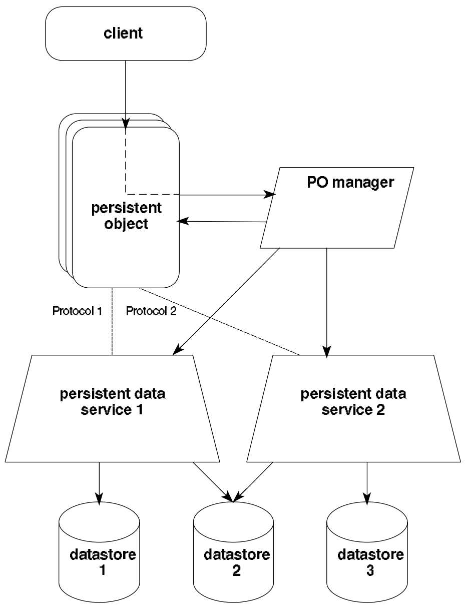

Figure 2 A possible OMG Persistent Object Service configuration

3 The weaknesses we see

While otherwise commendable, the effort of the designers to make the Persistent Object Service specification as open as possible may, in the final effect, somewhat limit the compatibility of compliant implementations. In certain conditions, these implementations may need to precisely define or otherwise specialize some aspects of the service and thus potentially render themselves incompatible with others facing the same problems. In the following, we mention those parts of the specification we consider most prone to such modifications.

3.1 Underspecified semantics of operations

Throughout the specification, the interfaces are designed "for a straightforward use". Handling of exceptional situations is missing. In our view, the important cases in this respect particularly include requests such as *disconnecting()* an object from a datastore location to which it has not been previously connected, *connecting()* several objects to one datastore location, or trying to *restore()* a state which is not compatible with the target object. Even if an actual implementation resolves these issues (most likely by considering at least some of such request as erroneous), it cannot report an exceptional situation back to its client through the standardized interfaces. Consequently, the implementation will introduce a proprietary solution to handling the situations not treated in the Standard.

3.2 Underspecified functionality of POM

Unjustifiably, the Standard does not delve into details of the Persistent Object Manager functionality. Contrary to the technical requirements of [OMG92b], the criteria used by POM to dispatch calls to attached PDSs are outlined very briefly. Again, vendors can be expected to introduce proprietary mechanisms for dispatching requests. For example, a potential dispatching mechanism might be based upon introducing user objects capable of providing the information necessary to route the request to the appropriate PDS (similar to the *TraversalCriteria* object defined in the Relationship Service [OMG94c]). In our opinion, this mechanism would have very little or no impact on the service flexibility. Moreover, a mechanism for dynamically registering available PDSs could be analogously introduced. Another dispatching mechanism might be for example based on making the identification of a target datastore a standardized public attribute of PID (in addition to datastore type).

In a more general view, it is not clear why *POM* and *PDS* are separate interfaces even though they provide syntactically identical operations. A more flexible approach could be based upon allowing for arranging PDSs into hierarchies (replacing thus *POM* by the root *PDS*). In fact, this would be similar to the concept of *GenericFactory* in the LifeCycle Service [OMG94a] which also benefits from such a recursive architecture.

3.3 Lack of "Compound Persistent Object Service"

One of the classical key issues of object persistency is the handling of inter-object references. The Standard, however, is mute in this respect. Furthermore, the POS architecture associates a persistent object only loosely with its PID, thus making the processing of inter-object references even more difficult. At the same time, we have found it surprising that there is no Compound Persistent Object Service as an analogy to the Compound Externalization Service and the Compound LifeCycle Service which define the cooperation of these two services with the Relationship Service.

3.4 Reusability of other Object Services

Another tricky issue in the Standard is the coexistence of the POS with other Object Services and reusability of other Object Services in the POS. Except for mentioning such an option, the Standard does not go into any details in this respect. In this section, we briefly analyze the prospects of reusing the Relationship, Externalization, and Naming Services. For details we refer the reader to [KPT96b].

3.4.1 Relationship

To handle relations among CORBA objects in a unified way, the Standard recommends using the Relationship Service. In the article [KPT96b], we showed that combining the Relationship Service with the POS raises several issues that must be solved, such as providing the client with a way to specify a *TraversalCriteria* object used for traversing a persistent object graph, or specifying the semantics of merging subgraphs when storing/restoring parts of a given graph. The necessity of resolving such issues makes reusing of the Relationship Service by the POS less straightforward than it might seem at first glance.

3.4.2 Externalization

Even though the Standard suggests the Externalization Service as a possible protocol, it does not go any further in specifying how the Externalization Service might actually be reused in the POS. In fact, as shown in [KPT96b], the Externalization Service can support the POS, inherently based on random access to individual objects, only in a very special case - when a POS implementation does not support fine-grained updating of parts of an externalized transitive closure of dependencies. As we also showed in [KPT96b], 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 using the remaining parts of the Externalization Service, thus not imposing the limit of not supporting fine-grained updating of parts of an externalized transitive closure of dependencies. (Such a protocol is also proposed in [Sess96] as we have found out recently).

3.4.3 Compound Externalization

As described in [KPT96b], combining the Compound Externalization Service with the POS introduces an unexpected conceptual mismatch. Basically, the Compound Externalization Service uses the *externalize_node()* method for externalizing a node together with all its roles. There may be cases when only a subset of all roles adjacent to the node needs to be saved, such as storing a subgraph. For such cases, the standard implementation of *externalize_node(),* which stores all adjacent roles, must be modified appropriately to avoid externalizing roles that have been left out by the *TraversalCriteria* object. The modification of *externalize_node()* logically demands changes in the implementation of its counterpart method *internalize_node()*. This method, which is according to the Compound Externalization Service specification responsible for loading all roles belonging to a node, would have to cope with internalizing subgraphs, choose a subgraph merging semantics, and also solve the problem of checking whether a particular role has already been internalized or not. The latter problem is especially hard since CORBA lacks means for checking an object identity (this functionality should have originally been provided by the LifeCycle Service, but has been omitted in the final version of its OMG specification [OMG94a]).

3.4.4 Naming

The Standard suggests using the Naming Service for translating human-readable names of PDSs into CORBA references. In our opinion, the Naming Service can also be used to provide mapping from human-readable names to PIDs. The level of indirection introduced by such a mapping may be beneficial for hiding the datastore-dependence of PIDs.

4 Conclusion

This contribution is based upon a thorough study of the Standard (originally without knowing the motivations published in [Sess96]) and our experience gained during an implementation of the POS for Orbix [ORBIXa, ORBIXb]. The main weaknesses we have identified include: underspecified semantics of the POS operations, underspecified functionality of POM, and very weak specification of how other Object Services may be reused in the POS. In addition, the Standard also lacks a section that would suggest means (by specifying additional interfaces or providing a corresponding semantics) for resolving the issues raised by combining the POS with other CORBA Object Services.

References

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

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

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

[KPT96a] J. Kleindienst, F. Plášil, P. Tůma: CORBA and its Object Services. Invited Paper, SOFSEM'96, Springer LNCS, 1996, to appear.

[KPT96b] J. Kleindienst, F. Plášil, P. Tůma: Lessons Learned from Implementing the CORBA Persistence Service, In Proceedings of OOPSLA'96, San Jose, Oct 1996

[MoZa95] T. J. Mowbray, R. Zahavi: The Essential CORBA, J. Wiley & Sons, 1995.

[OHE96] R. Orfali, D. Harkey, J. Edwards: The Essential Distributed Objects. Survival Guide. John Wiley & Sons, 1996

[OMG92a] Object Service Architecture, OMG 92-8-4, 1992.

[OMG92b] Object Services Request for Proposal 1, OMG 92-8-6, 1992

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

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

[OMG94c] Relationship Service Specification, Joint Object Services Submission, OMG 94-5-5, 1994.

[OMG94d] Compound LifeCycle Addendum. Joint Object Services Submission. OMG 94-5-6, 1994.

[OMG94e] Object Externalization Service. OMG 94-9-15, 1995.

[OMG95a] Common Object Request Broker Architecture and Specification Revision 2.0, OMG 96-3-4, 1995.

[OMG95b] 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.

[Sess96] R. Sessions: Object Persistence, Beyond Object-Oriented Databases. Prentice-Hall 1996

[Sie96] J. Siegel: CORBA. Fundamentals and Programming. J. Wiley & Sons, 1996