CORBA Persistence Object Service using PerDiS

Edi Ray Zavaleta Olea, João Garcia, Paulo Ferreira
Distributed Systems Group, INESC ID LISBOA / IST
Rua Alves Redol 9, 1000 LISBOA, Portugal

Abstract
There has been research on syntactic level extensions to the Common Object Request Broker (CORBA) Interface Definition Language (IDL) in order to obtain persistence for CORBA application objects (e.g. CORBA Persistent Object Service). The purpose of this paper is to present an implementation of the CORBA POS, which provides efficient persistence and improved access to complex objects, using PerDiS. PerDiS is a persistent distributed store that provides applications with a distributed shared memory abstraction, and supports transactions, garbage collection, security and object caching on both local and wide-area levels.

Keywords: Caching, CORBA, Persistent Distributed Store, PerDiS, Data-shipping

1. Introduction
The development of distributed applications requires support for persistence between heterogeneous distributed object systems. Given that uniformity of platforms and systems software is not possible in practice, a uniform middleware abstraction is a viable alternative to unify the several programming interfaces offered today. Middleware architects and vendors of software propose their own architectural standards, e.g. OMG (Object Management Group) CORBA (Common Object Request Broker Architecture)[10][18], Microsoft DCOM (Distributed Component Object Model), Sun Microsystems JavaBeans, etc. CORBA is a standard proposed by OMG [15], which unifies object programming, and distributed systems technologies in order to support interoperability and heterogeneity between objects store.

Existing implementations of CORBA brokers suffer from performance problems. These include the communication latency inherent in distributed systems and overheads associated with message exchanges when a CORBA client invokes methods of an object that is implemented by a remote persistent object store, in particular if using function-shipping. These overheads can be reduced significantly if client applications could access the objects’ state locally, which could lead to better performance. Previous work has demonstrated the benefits of caching in a range of systems from Distributed Shared Memory (DSM) to distributed stores. The caching technique and its implementation have not been addressed in the CORBA specifications. Our goal is to provide client CORBA application with an efficient access to persistent object located in a Distributed Persistent Object Store (DPOS) using caching and data-shipping techniques. DPOS often support sophisticated features such as long running transactions, complex concurrency control, object caching and other adaptations to distributed (even wide-area) environments. There are several research prototypes of such systems (PerDiS [6], Thor [18], etc...).

We would like to provide a comprehensive architecture and a set of development tools, to provide integration, automation, efficiency, and easy-to-use between CORBA applications and PerDiS stores. Application programmers should be able to concentrate on application semantics. In co-operative engineering applications, which are the type of applications we want to support, object servers may be located across a slow WAN connection, that require efficient persistence, support of very “fine-grained” persistent object and adding object caching to improved performance. For example, in virtual enterprises (a temporary consortium of several companies), where small groups of users share numerous objects asynchronously and perform many updates during a limited period of time, using object caching and data-shipping could improve the performance of distributed object systems.

In this paper, we present a design for allowing CORBA applications to store and efficiently access objects in a persistent distributed store. It enables the specification of CORBA applications using the Persistent Object Service, which, with automatic code generation, can encapsulate the complexity of distributed application and of object caching protocols (see Figure 1). In the next section, we present Related Issues and Previous Work. Section 3 describes the conceptual architecture. Section 4 presents a sample application. Section 5 presents the results of performance tests regarding method invocations of persistent object between CORBA vs. CORBA-PerDiS. They demonstrate that caching could result in improved performance for applications that frequently access these objects. Finally Section 6 presents the paper’s conclusions.

![Figure 1 - CORBA and Persistent Distribute Stores](image_url)

2. Related Issues and Previous Work
In order to support persistence, CORBA provides a collection of object services called Persistence Object Services (POS) [14] (see Figure 2). They were specified by OMG and define the interfaces to data and persistence object using the Interface Definition Language (IDL; for example, see Figure 3). The main objective of POS is to allow “uniform interface to many different data stores” [7]. CORBA POS is composed of:

- **Persistent Objects**: An object with a Persistent State (PS) whose lifetime can exceed the lifetime of the application it is used in. The PS is used to guarantee the persistence of an object independently of application execution (session...
state, etc...) and of failures, such as shutdowns, systems crashes, etc. Client application access persistent objects using PID, which is a unique identifier that describes the data store location of the persistent object.

- **Persistent Data Service**: These are interfaces to access particular data store implementations. Their functionality is to move data between an application object and a data store.

- **Persistent Object Manager (POM)**: Dynamically resolves the binding between PO’s and Persistent Data Service (PDS), using the persistent object’s PID and the protocol(s) supported by it.

- **Data Stores**: A variety of persistence services -- SQL Databases (SQL DBs), Object Data Base Management Systems (ODBM), Simple Object Stores.

![Diagram of the OMG Persistent Object Service](image)

**Figure 2 - Components of the OMG Persistent Object Service.**

Existent prototypes have introduced several research solutions to the problem of object persistence, by providing the IDL CORBA (for instance shown in Figure 3), that use implementation-specific POS protocols for data exchange between an object and PDS. Among the most important are:

1. The Direct Access (DA) protocol is SunSoft proposal for directly accessing persistent object using OMG Persistent Object Service. The DA protocol uses an IDL-like Data Definition Language (DDL), to define persistent data. The designer of a CORBA object implementation uses DA DDL, to describe the object’s persistent state in one or more data object interfaces. The DA protocol does not support fine-grain concurrency access to persistent data [5].

2. The ODMG-93 (Object Data Manager Group): Persistence efforts from the database community are grouped within the Object Data Manager Group [10] that proposed the ODMG-93 standard. ODMG-93 it is an extension of CORBA POS for provides a protocol with efficient Persistent Data Services (PDS). Its main characteristics are:

   - Provides an Object Query Language (OQL) ODMG-93, which supports queries in SQL and transactions.
   - The ODL (Object Definition Language) ODMG93 provides high-level primitives for the data definition of different collections of objects (set, bags, lists, etc...).
   - ODMG-93 defines how to write C, C++ Smalltalk or code that manipulates persistent object.

In the multidatabase area there is MIND (METU Interoperable DBMS) [3][4], that it is a multidatabase system (SQL DB’s and ODBM) over CORBA. The integration of export schemas is performed using ODL, which is based on CORBA IDL. MIND provides the query data using a global query language based on SQL, supports a global transaction manager with a global serializable execution of transactions. Some of the most important products that provide extensions to the IDL using ODL/OML are:

   - Secant's Persistent Object Manager (POM): it provides a group of tools that compose business objects CORBA with a variety of relational database technologies [16].
   - TERSOL offers an ODBMS and a programming environment that enables the definition and manipulation of an object database model using high-level ODL. It automatically produces a set of C++ classes for that model and seamlessly composes persistence into C++ applications [19].
   - OpenORB provides a CORBA framework, using the Persistent State Service specified by O.M.G, for manipulate persistent data in relation database [13].

```
module StoreAccess{
  struct state_object{
    long key;
    char firstName[80];
    char middleInitial[80];
    char lastName[80];
    char location[80];
    long phone ;
  };

typedef sequence <state_object>
infodata_Seg;
... };

module Bank{
  interface Accounts{
    ...
    boolean add_Account(in string key, in StoreAccess::state_object data);
    any display_Account(in string key);
    any select_Accountl(in string key, out long size);
  };
}
```

**Figure 3 - Bank IDL: CORBA IDL Definition of the Bank Interface.**

Considering the different alternatives for supporting persistence using the extended IDL between CORBA and database technology, the most adequate proposal that has appeared during last decade has been the introduction of OMG POS, which recog-
nized that absence of a viable CORBA specification for persistence. However, this work has several problems:

- **CORBA** does not support relationships well and most corporate object models are very complex in nature and highly interconnected.
- **CORBA POS** does not specify how objects in DPOS, which meet particular application domain requirements, can be accessed.
- It does not address the issues of store management (naming, access control, open/closing), access using data structures (C++ classes retrieval) or memory representation of persistent objects.
- Its persistent object service does not present an implementation that supports a protocol for data exchange between a CORBA object and DPOS.
- The languages proposed by OMG Persistent Object Service, ODMS-93, which is composed by ODL/OML, specify only the logical characteristics of objects and the operations used to manipulate them in ODBMSs.

### Problems with efficient persistence and data-shipping

In order to efficiently support persistence and data-shipping between CORBA and PerDiS, we had to solve two main problems: extending the standard IDL grammar (without OMG Persistent Object Service) with the concepts needed to use a PerDiS store, and establishing a mapping between this subset of the IDL (for example, Figure 6) and the PerDiS API code. These problems can be detailed as follows:

**Syntax:** There should be a syntactic equivalence between the public methods of the POS protocol and the state management of persistent objects in PerDiS. The main hurdle is mapping data types: We use the IDL data type definition so that persistent objects can be stored as collections, which the programmer defines explicitly. In PerDiS, these collections are mapped onto C++ Standard Template Library (STL) types (e.g. list, set, bag or array) [17].

**Semantics:** We need to define how applications interact with persistent objects and the store, the semantics of object invocation, the instantiation policies, etc... The main aspects of this problem are:

- **Interfaces:** CORBA interfaces were extended with primitives which: a) Ensure that client invocations of an object are performed within a PerDiS transaction. Transactional actions performed by objects are beginning a transaction (begin transaction) and terminating a transaction using a commit (commit) or abort (abort). In the case of commit failure (causing an abort), transactions are rolledback. b) Locks: used for synchronizing access to persistent objects. Given that several replicas of same object, and concurrent access to them, may exist at a given time, it is necessary to support primitives for object locking (lock) and the model according to which it is done: explicit vs. implicit, optimistic vs. pessimistic. Implicit locking is an additional locking policy available in PerDiS, where applications need not lock objects via PerDiS API calls. The detection of data access is made using memory protection and page faulting. When an application accesses a protected memory page, data is fetched from the store the page is unprotected and added to the transaction’s read and write sets.

- **Communication Protocol:** In order to formulate or exchange messages between PDS and PerDiS. These require a communication channel and a protocol for the interpretation of the messages. Other communication requirements can be partial ordering of messages or concurrency control.

- **Cluster Management:** Mapping between the Distributed Shared Memory representation of distributed objects and PerDiS’ cluster management involves naming, cluster access, using data structures (C++ classes retrieval). The cluster in which states objects are stored is created using the cluster name and the class’ data type.

- **Persistent Data Service:** This is used to traverse the persistent store and access the objects’ state. It resorts to C++ STL algorithms. These can work with abstract data types and the STL iterators allow the traversal of CORBA-PerDiS clusters. We use the `end` operator of STL containers as the point to insert new objects. Object query and state display is done using iterators over object containers and by testing the objects’ PID for the correct identity.

### 3. Conceptual Architecture

We propose a comprehensive architecture to provide CORBA POS using PerDiS. This architecture includes the application client (CORBA) and the object implementation (Class Code) using POS. It provides transparent persistence by mapping between memory virtual address space and the persistent store using file mapping. The definition and specification architecture can be represented symbolically as follows:

\[
\text{InterPersist Architecture} = f(\text{Uniform Interfaces, Persistence, Performance, Consistence, Scalability}).
\]

**Persistent Distributed Store (PerDiS)**

PerDiS offers an easy to use middleware platform. PerDiS supports a shared address space model by means of transaction systems running on top of a DSM layer. It provides support for distributed collaborative engineering applications [6]. Each PerDiS node has a server called PerDiS Daemon (PD) and applications that interact on a client/server basis with the local PD. PD’s are organized in a peer-to-peer structure. Client applications are linked with a User Level Library (ULL) and interact with the system using an API (PerDiS Application Programming Interface). The ULL mainly deals with application-level memory mapping and management of clusters of objects, and communicates with the local PD, which communicates with other PD’s over the network.

PerDiS takes advantage of the Shared Address Space Model [5] to implement the direct access approach to access PO’s and to support Persistence by Reachability (PBR). This abstraction facilitates transparent and coherent distributed-shared memory with an in memory location at which the object is stored. PerDiS overcomes the remote invocation bottleneck by local replication of all accessed data. Each object has a unique identity, and a set of methods. Persistence objects that are reachable from named objects, called persistent roots, are persistent; object that are not accessible from persistent roots will be garbage collected and removed. Objects are grouped in collections called clusters, where a cluster is an abstraction for a set of files called bunches. A cluster has properties of a heap because a program allocates an object in a specific cluster, and properties of a file because it has a name and attributes. As an example, **Figure 4** shows PBR between two clusters located at two different servers.
Components

We propose a development environment (see Figure 7) based on a set of development tools called InterPersist. It takes the extended CORBA IDL, called IPOL (Interoperability Persistent Object Language) and a precompiler which automatically generates the necessary code for create a storage home in PerDiS store. This includes data type mapping between both systems and the persistence (PerDiS) related parts of the CORBA server classes. IPOL additionally, links the automatically generated code in application development to ease the development process. The IPOL language is formed by State Object Definitions. This part of the language allows programmer to declare the name data type of state object (storage_type), name and type collection (sets, bags, list) that they want to use with POS. The POS has a method that creates a storage home and a STL con- tainer within this storage homes where its instances will be stored. A mapping performed by the InterPersist precompiler, translates the definition in IPOL into standard C++ STL supported by POS.

Development Approach

The application development process has a set of programs to support the definition of object states and collections used in IPOL. The programs use IPOL compilers to support automatic code generation.

The application development approach (shown in Figure 7) supports the following phases:

- Declare the class prototype in CORBA IDL, which includes the definition of name and signatures of operation between objects (syntactic-level).
- Run the IDL file through a standard C++ CORBA precompiler. The IDL compiler processes the IDL files generating Client/Server Stubs.
- Add service semantics in IOPL to the generated storage homes. This allows adding IOPL semantic specification to the IDL, and describing the usage and the capabilities of the state object into PerDiS.
- Run the IOPL precompiler. It generates code for the C++ implementation of the DPS classes, and C++ headers for the defined storage classes and STL collections.
- Compile the C++ code. This task consists of compiling the CORBA/PerDiS C++ generated code and linking it with the CORBA object broker run time library and the PerDiS ULL.

```cpp
interface COProtocol {
    boolean save (in StoreAccess::state_object data, in string PID);
    boolean load_state (in string PID, out StoreAccess::state_object data);
    boolean delete (in string PID);
}
```

The final results of the code generation are:
- C++ header files: Interface definitions for the classes’ methods and STL definitions for storing classes in a PerDiS store.
- C++ class code CORBA/PerDiS: The generated class code includes the PerDiS API calls within the CORBA implementation of the interface seen by the application.
- IPOL run time: InterPersist has a C++ class library that is linked with the CORBA server classes and which implements basic PerDiS persistence operations.

![Figure 5 - Architecture of InterPersist environment](image)

To access an object’s persistent state we use the POS protocol methods for manipulating clusters (see Figure 6). The methods presuppose the following rules: a) Each method call is a transaction. b) If the user does not specify how objects are locked, implicit locking is used. c) Locks are held until the transaction ends.

![Figure 4 - Remote reference in PerDiS](image)

4. Sample Application

The account example is a simple application to create a bank account using IPOL. The code fragment in Figure 8 shows the definition of the object’s metadata. The type of container used to persistently store instances of the account class is specified.
by the seg_type keyword. IPOL containers are mapped onto STL collection types and C++ classes.

```cpp
// In file people.iopl
abstract storage_type Account;

abstract storeseq Bank {
  seqname = "state_object";
  seqtype = "list";
};
abstract storehome Person {
  type = "pds";
  hostname = "hoare.cic.ipn.mx";
};
```

Figure 8 - Bank IPOL: Object definition and creation of the people store.

InterPersist automatically generates C++ code for the CORBA-PerDiS classes based on the IPOL definitions. The generated code is formed by C++ header files and C++ class code for the CORBA-PerDiS object model. For example, a source code fragment demonstrates (Figure 9) the use of SOO with PerDiS.

```cpp
CORBA::Boolean
Bank::Accounts_i::add_Account (CORBA::String ind, const StoreAccess::state_object & data) {
  ...  
  if (a >= str) {
PO->save_state(data, a);
  }
  ...  
}  ...  
CORBA::Boolean Bank::Accounts_i::
delete(CORBA::String PID)
{
  ...  
  PO->delete_state(PID);
  ...  
}
```

Figure 9 - BankImpl: C++ Class CORBA Bank Implementation Code.

## 5. CORBA vs. CORBA-PerDiS performance comparison

This section evaluates the performance of persistence and object invocation speed in CORBA vs. CORBA-PerDiS. In tests, we ran a client/server program in different situations to compare the performance of C++ CORBA and CORBA-PerDiS static method invocations on persistent objects. The client program performs 10000 methods invocations during the elapsed time. The test environment is composed of the following machines:

- 166 MHz Dual Pentium, 64 Mbytes RAM running NT 4.0 Workstation with 100 Mbytes of swap space.
- 333 MHz Pentium II running NT 4.0 Workstation with 100 Mbytes of swap space.

These machines are connected across a 100 Mb/s Ethernet LAN and are configured with PerDiS and OmniORB2 broker version 2.7.1 [12]. One machine was designated as a server (b) and the other as a client (a).

![Figure 10 - Test application execution time on a local server.](image)

In the first test, we ran client and server programs on different machines using CORBA function shipping to access objects, stored in a file on disk, which are served by an ORB. In the second test, we ran a CORBA client compiled with InterPersist that used PerDiS to accessed a PerDiS store on another machine. CORBA is more efficient for small objects, which are invoked a small number of times. This is due to the fact that the initial access to a PerDiS object is expensive due to the cost of a PerDiS transaction. As this cost gets diluted in the complexity of the work that is performed, represented in our test by invocations to an object, PerDiS becomes more efficient than CORBA. Furthermore, PerDiS takes advantage of DSM and caching to avoid having to call the remote server.

![Figure 11 - Test application execution time on a remote server.](image)
6. Conclusions

The main result presented in this paper is an approach for implementing CORBA POS efficiently using PerDiS. PerDiS supports persistence, transactions, locking, handling error and object caching for “fine-grained” objects with complex C++ structures. We present a comprehensive architecture, InterPersist, which provides a set of development tools that integrates the CORBA and PerDiS infrastructures.

InterPersist provides programmers with IPOL, an extension to the CORBA IDL, which can handle the complexity of mapping CORBA objects into PerDiS and automatically generates the necessary code, thus reducing the development time of C++ code for a CORBA/PerDiS application. This allows programmers to write in a single integrated programming language ensuring the development of application using extended CORBA IDL. IPOL provides a syntactic and semantic description of object classes whose instances are persistently stored in STL collections and transparently accessed using distribute shared memory.

This approach presents an alternative to previous CORBA persistence experiments. The properties of PerDiS (e.g. data-sharing, object caching and DSM) provide an efficient and scalable solution for persistence for CORBA applications although the costs of supporting transactions can be high for simple applications. However, InterPersist is a good solution for porting CORBA applications that are aimed at a distributed collaborative engineering environment.

Furthermore, these overheads are reduced significantly if client application access the object state locally repeatedly, allowing subsequent invocations to be executed locally which leads to better performance. In the future, we would like to include language extensions to IOPL, which support information exchange with persistent object stored in traditional or CORBA POS databases.

7. References


