Flexible Persistence Framework for Object-Oriented Middleware

Mathias Weske*, Dominik Kuropka†

June 25, 2001

Abstract

The ability to store object states persistently in a convenient and flexible way is an important requirement for middleware environments. In this paper we propose a language independent framework for flexible and extensible persistent storage services for object-oriented middleware. The framework is specified by interface definitions and the rationale of their interplay. To support application-specific requirements like particular data repositories and distribution aspects, different persistence services are feasible within the proposed framework. These services are easy to use, since objects are stored persistently by inheriting from a single class; additional auxiliary classes–like, for instance, factory or session classes in related approaches–are not required. We remark that inter-object transactional behavior is not in the scope of a persistent storage service; these issues are addressed by transactional services. To show the open design of the proposed framework, we sketch how the framework can be integrated with the Transaction Service proposed by the Object Management Group.

Keywords: Persistence Framework; Object-oriented Middleware; CORBA

1 Introduction

The ability to store arbitrarily structured objects in persistent storage in a flexible way with little effort is an important requirement for enhancing the reliability and fault tolerance of object-oriented middleware [13, 14, 5, 20]. If the representation of a persistent object in volatile storage is lost, e.g., due to a server crash, the persistent storage service is responsible for restoring a consistent object state after the system restarts. Recovery from failures should be transparent to the user. In addition, application programmers should concentrate on application-specific properties and leave issues related to the persistent storage of object states to the persistence service. In this paper

*Contact Author: Mathias Weske, Hasso Plattner Institute for Software Systems Engineering, Am Luftschiffhafen 1, 14471 Potsdam, Germany. Phone: +49 331/90972-61, Fax: +49 331/90972-10, E-Mail: weske@hpi.uni-potsdam.de

†Contact Author: Dominik Kuropka, University of Münster, Department of Information Systems, Leonardo-Campus 3, 48149 Münster, Germany. Phone: +49 251/83-38079, Fax: +49 0251/83-38109, E-mail: isdoku@wi.uni-muenster.de

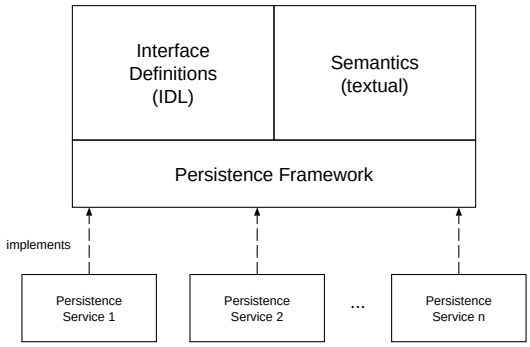

Figure 1: Persistence Framework versus Persistence Services.

we propose a flexible, extensible, and language independent persistence framework for object-oriented middleware.

The framework is specified by interface definitions. Implementations of the framework result in persistence services that are used to store application objects persistently. We stress that within the framework, a variety of implementations are feasible to support different application requirements, for instance different types of data repositories (e.g., file systems, relational and object-oriented database systems), data representations (e.g., defined by binary streaming or complex schema mapping). In addition, the framework is language independent, meaning that all programming languages supported by the middleware can be used to implement persistence services based on the proposed framework. It is important to remark that different implementations of the framework are transparent for the users of the service, i.e., application objects. The relationship between the proposed framework and particular implementations is shown in figure 1: The persistence framework is specified by interface definitions using Interface Definition Language (IDL) and by their semantics, which is described textually.

We like to clarify the following point: In this paper we propose a framework for a persistent storage service, not a particular implementation. Since the term framework is often used (and sometimes misused), we state that we mean by framework the specification of a set of interfaces as well as the specification of their meaning and interplay. To this end, we discuss in some detail the objects involved in persistent storage, what these objects are supposed to do and how they interact to reach their goal of persistently storing states of application objects.

This paper is organized as follows: In Section 2, the application model is characterized, and we sketch aspects not covered in this paper in detail. Section 3 introduces the persistence framework by presenting its design goals, its overall structure including the classes involved, the interface definitions of these classes and their interplay. Implementation aspects of a particular persistence service based on the proposed framework are discussed in Section 4. A section on related work and concluding remarks on an integration with the Transaction Service specified by the Object Management Group (OMG) and remarks on some optimizations complete this paper.

2 Preliminaries

This section introduces the environment of the persistence framework, and it characterizes the scope of this paper by sketching issues not addressed.

2.1 Application Model

In object-oriented systems, applications are designed and implemented by objects, which are software representations of real-world entities like customers, orders, and invoices. Objects are characterized by structure (a set of attributes) and behavior (a set of methods with signatures attached). Objects communicate by message passing. Once an object receives a message, it determines whether the respective method is available, in which case the method is invoked and the return values are sent back to the calling object. If there is no such method, the system raises an exception.

In distributed computing environments, objects of a given application may reside in different computing nodes. As a result, method calls are nonlocal in general. This means that an object in site A may send a message to an object in site B, and the latter object returns the result of the method invocation to the calling object. The software system that handles the communication is called middleware. Object request broker are a well known and widely used middleware that is based on the Common Object Request Broker Architecture (CORBA) specified by OMG. In CORBA environments, messages between objects are transferred by an object request broker. Object locations are transparent to application objects, which means that calling objects do not need to know in which site the requested object is located. As a result, site changes of objects do not require modifications of the application: CORBA object identifiers are resolved to sites by the middleware.

The persistence framework proposed is conceived for these environments; it provides a convenient and flexible way to store object states persistently, with the aim of enhancing the fault-tolerance of object-oriented applications. Generally speaking, at each instance, an object is characterized by its state and its behavior. While object behavior is specified by methods, the state of an object is represented by the object's current attribute values. Consequently, a persistent storage service deals with storing attribute values persistently, with the aim of enhancing fault-tolerance and availability. It is designed for client/server applications with special focus on business applications in heterogeneous environments. Given the myriad of approaches to add persistence to Java (the most relevant ones are discussed below), our approach is language independent. In fact, the specification can be implemented in C++, C, Java, or other programming languages that are supported by CORBA.

There are two groups of persons who make use of the results presented in this paper. Firstly, the system designers and system programmers who use the proposed persistence framework to implement specific persistence services. For instance, classes representing database objects have to be implemented for each database system to be used to store object states. The second group of persons are application developers, who use the persistence services during application development. For instance, in an order processing application, the application class order consists of a variety of order objects. To store these objects persistently, application programmers call a method

provided by the persistence service. Notice that any changes to the implementation of the services is transparent for the application programmers, and hence, transparent for the application. For instance, it may be required to change the data repository from a file system to a relational database system. This can be done conveniently by modifying the persistence service implementations. Applications, however, do not have to be modified at all.

2.2 Aspects not Covered

Since object persistence is related to a variety of concepts and techniques, this section discusses the issues that are not addressed, namely inter-object transactional properties and performance evaluations. However, since the service is extensible and open to cooperate with other services, this does not restrict the usage of the proposed persistence framework. In the contrary, by its limited scope, the specification and implementation is rather light-weight.

An important issue related to persistence is transactional support for object-oriented applications. The transaction concept is well-known from database systems, where concurrency control and recovery issues have been addressed for some time, generating theoretical and practical solutions of considerable impact [2, 7]. In general, concurrency control deals with synchronizing concurrent accesses of a set of transactions to shared data, stored in a database. Important problems in this context are dirty read, lost update, and inconsistent retrieval [2]. It can be shown that these issues can be solved with locking protocols. In transaction processing terminology, locking protocols generate executions that obey certain correctness criteria, for instance, conflict serializability. It is worth mentioning that transactional issues and persistence issues in object-oriented programming are rather independent. In particular, concurrently active objects invoking methods on each other will run into lost update problems if transactional capabilities are not present to control their interaction. Hence, transactional issues may occur even without any persistence. On the other hand, objects in volatile storage may be lost due to the loss of main memory content, which is a problem that persistence support has to handle — transactional properties of objects are not involved in this case. While this paper focuses on persistence, it also discusses how the proposed persistent storage service can be integrated with a specific transaction service (cf. Section 6.1). However, on the storage level, transactional support is required for the proposed service. Object states are stored in persistent storage transactionally, i.e., each object is either written completely or not at all. Hence, transactional properties of the underlying data repository are used in the storage level. However, inter-object transactional behavior is not in the scope of a persistent storage service and therefore, consequently, is not addressed in this paper. We remark, however, that a transaction service can make use of the persistent storage service, as is discussed in some detail in Section 6.1.

Since the main objective of this paper is to introduce the conceptual design of an extensible persistent storage service, we do not provide a performance evaluation. Performance can only be analyzed for particular implementations. As will be discussed below, an interesting aspect of the proposed service is the ability to cover a variety of different implementations, in the sense that within the proposed specification, imple-

mentations focusing on different aspects can be provided. For instance, integration of legacy applications and support for large, unstructured multimedia objects like audio or video data. We mention, however, that the performance of the proposed service is effected by the storage policy used, which is discussed in more detail below. In addition, Section 6.3 presents some ideas to optimize the service.

3 Persistence Framework

The conceptual design of the persistent storage service is based on a set of design goals, discussed below. The conceptual design of the persistent storage service is presented by its general structure and the interfaces of the classes involved. An example shows the usage of the service.

3.1 Design Goals

Since the main goal of this paper is the conceptual design of a persistent storage service and not its implementation, there are distinct design goals for implementors of the service and for users of the service. Implementors are persons involved in developing code to implement the service. Users of the service are typically application programmers, who make use of implementations of the persistent storage service proposed. The following goals were identified for the conceptual design of the persistent storage service.

- *Ease-of-Use:* For efficient software development and maintenance, application programmers, i.e., users of the service, should concentrate on the business logic, rather than on technical details like the persistent storage of application objects. Incurring considerable additional coding to store application objects persistently is not an attractive feature of a persistent storage service. In particular, it seems inappropriate to require application programmers to develop additional interfaces and classes for providing persistence for application objects. In contrast, providing methods for storing object values persistently and for loading object values from persistent storage in volatile storage is a more appropriate choice.

- *Class Evolution:* Given the dynamic nature of todays business environments, the structure of application objects is likely to change due to new requirements imposed by the application. For example, objects may require new attributes or given attributes are no longer needed. In the former case, the respective attribute values have to be stored persistently to properly represent object states. As a result, it is an important design goal for users of persistent storage service that evolution of both classes and objects is supported.

- *Data Integrity:* In presence of system failures involving the loss of volatile memory content, the system should be able to recover data from persistent storage, i.e., to load persistent data into object attributes. Data integrity and consistency have long been studied in the context of database systems, where the transaction concept was introduced [7]. In the context of a persistent storage service,

writing objects to persistent storage and loading objects from storage should be performed transactionally, i.e., either completely or not at all.

- *Data Repositories, Distribution and Heterogeneity:* In today's organizations in commerce and public administration, typically a variety of data repositories are present, due to evolving information system infrastructures. Consequently, it is an important requirement of a persistent storage service that objects can be stored in different data repositories. A persistent storage service that is limited to a single data repository (or that requires considerable coding when it comes to using an alternative data repository) is not an attractive choice. In addition, changing the underlying data repository should be possible without changing objects in the the application level. In contrast, it should suffice to change the implementation of the persistent storage service.

  Distribution and heterogeneity are also found in most real-world information system infrastructures. Hence, a persistent storage service should allow to store objects in different data repositories that may even belong to different types, for instance, relational databases, POSIX file systems, and object-oriented database systems. Since these different data repositories are typically run on different computer systems, distribution aspects also have to be supported. In particular, it should be possible to store objects in different data repositories that run in different machines in a distributed fashion.

- *Data Representations:* Different applications may require different data representations of object values in persistent storage. For instance, when existing domain-specific application systems (legacy systems) and object-oriented applications access a common data repository, care has to be taken in mapping attributes of persistent objects to structures of the data repository. For example, if the legacy application stores data in a relational database table, then mapping attribute values to columns of existing tables in the relational database is required.

  Different data representations within a given interface definition is also useful for software development projects. During prototyping phases it may be reasonable to store objects in binary format, which can be implemented simply by streaming object states to persistent storage. While the semantics of the persistent object's attributes are lost in this representation (the complete object state is represented by large binary object with no internal structure), this data representation is easy to implement, which may enhance the efficiency of software development projects.

- *Storage Policy and Granularity:* When dealing with persistence in object-oriented applications, the question on when to write object values to persistent storage is crucial from a performance point of view. Performing write operations for each manipulation of an object's attributes can amount to a considerable overhead, if there are many modification operations of individual object attributes. On the other hand, performing write operations only after long periods of time have elapsed may result in corrupted data, since the data in persistent storage does in general not reflect the current object state. Depending on the

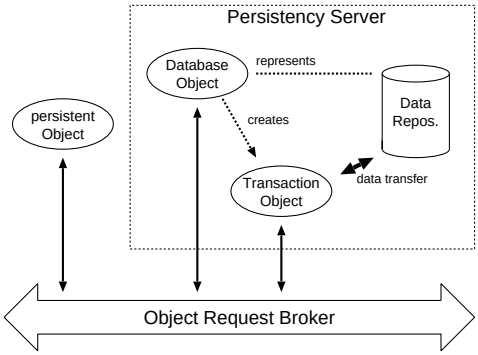

Figure 2: General Structure of Persistence Framework

requirements of the application, the persistent storage service should provide the flexibility to support different storage policies. In particular, it is advisable to support the writing of partial objects, i.e., only of attributes of an object that really have changed since the object value was last made persistent.

We remark that implementors of the service can make use of distributed, heterogeneous data repositories and storage policies to develop efficient and flexible service implementations, while these internals of the service implementations are transparent for users of the service, i.e., for application programmers.

3.2 General Structure

The general structure of the persistence framework is shown in figure 2, where a persistent object and the relevant objects of the persistence service are displayed. The persistence service is implemented by a persistence server that contains objects representing a data repository and a transaction, respectively. In particular, a database object represents a data repository, and a transaction object is created by a database object. Communication between the objects is facilitated by the object request broker. The internal structure of the objects involved and their interplay described by the following interfaces:

- *Interface PersistentObject:* Each persistent object inherits from the PersistentObject interface that contains the attributes and methods to store objects persistently. To fulfill this task, a persistent object communicates with database and transaction objects within the persistence server using the object request broker.

- *Interface Database:* The Database interface represents data repositories. In particular, for each data repository used, a database object is created. Hence, incorporating new data repositories amounts to creating new objects of the Database class. We remark that for each type of data repository, an implementation of the Database interface is required to capture the specific properties of the system used. As a part of a persistence server, each database object represents a

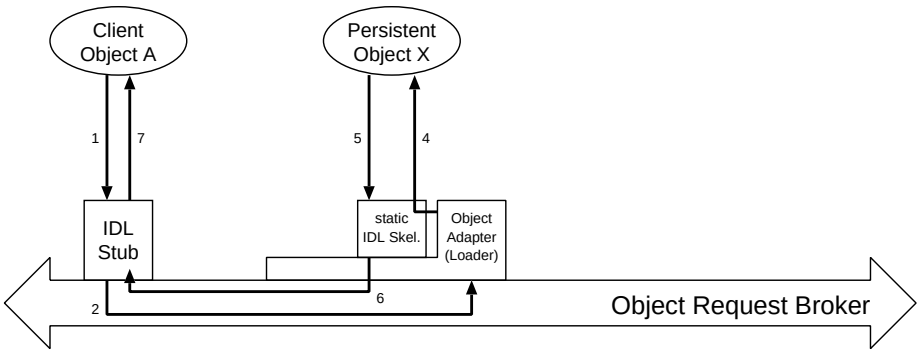

Figure 3: Processing a CORBA Request.

data repository and implements methods for accessing the data repository, for instance by providing methods for the creation of transactions.

- *Interface Transaction:* The interface Transaction represents transactions as atomic units of work; in particular, accesses to the data repository have to be executed transactionally. Just as database objects represent data repositories, transaction objects represent database transactions. Again, for each type of data repository, there has to be an implementation of the Transaction interface, to capture the specific properties of data access in the data repository used.

  Transaction objects are part of the persistence server. They implement methods to store and restore data from the data repository represented by the database object that created them. Persistent objects store (restore) attributes in (from) the data repository by communicating with transaction objects.

The interplay of the introduced interfaces is explained by the two main tasks the framework has to fulfill: storing objects in persistent storage and loading values from persistent storage.

Requests to CORBA objects are processed as shown in figure 3. The request is triggered by an object A that calls a public method of a persistent object X. In CORBA environments this means that A executes a local request on the representative IDL stub of X. The occurrence of this event is marked by (1) in figure 3. We assume that the requested object X is in volatile storage. In this case, the middleware marshalls the request and transfers the request to the object adapter (2) and finally to object X (4). X processes the request and transfers the return values via the static IDL skeleton (5) to the IDL stub (6) and to client object A eventually (7). Notice that this is the default behavior of CORBA calls.

If, however, persistent object X is not in volatile storage, the current object state of X has to be loaded from persistent storage. The steps involving the processing of this request are shown in figure 4. Steps (1) and (2) are equivalent to the previous case. When processing the request, however, the object adapter figures out that object X is not in volatile storage. As a result, it passes the request to a loader. The loader functionality can be provided, for example, by the Portable Object Adaptor, as detailed,

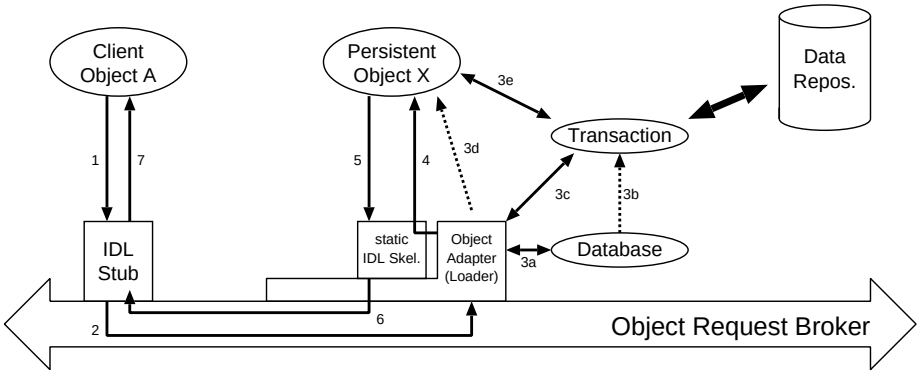

Figure 4: Loading Object State from Persistent Storage.

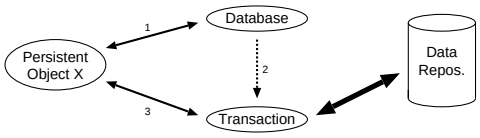

Figure 5: Storing Object State into Persistent Storage.

e.g., in [16]. The loader tries to re-initialize the object by contacting the database (3a) and creating a transaction (3b). Using the transaction object, the loader determines the structure of the persistent object (3c). Knowing the structure and class of the persistent object X the loader is able to create the object in volatile memory (3d). At this instant, object X is in the initial state. Due to the object-oriented approach, object X is responsible for restoring its actual state. To do so, X uses the transaction object to restore itself (3e). In the next step, the actual attribute values are restored in X, the middleware component sends the original method request to the now restored object X (4) that performs the method and returns the values via the middleware (5, 6) to the calling object (7).

The process of storing persistent object X is shown in figure 5: In the first step the object contacts the database (1) to obtain a transaction on the data repository (2). X uses the data access methods of the transaction to store all of its attributes (3). Finally the transaction is commited and closed by X. This discussion describes the general structure of the storing and loading, respectively, of a persistent object; the interfaces of the classes involved are discussed in some detail next.

3.3 Interface PersistentObject

The class PersistentObject is the main class of the persistence framework. An excerpt of its interface is shown in figure 6. In order to make the objects of a class C persistent, C inherits from the PersistentObject class. Obviously, C then inherits the behavior with respect to persistence from the PersistentObject class. Each persistent object has

a dedicated data repository as a default storage location. The reference to this object is returned by the getDatabase() method of the PersistentObject class. To load and store object states, the PersistentObject class has the methods saveMe(), saveMeT(), loadMe(), and loadMeT(), which are discussed in turn.

```
module CosPersistency {
  interface PersistentObject {
    void saveMe(in boolean undo_when_failed)
           raises(Failed);
    void saveMeT(in Transaction t) raises(Failed);

    void loadMe() raises(Failed);
    void loadMeT(in Transaction t) raises(Failed);

    Database getDatabase();
  }
  interface Database {...};
  interface Transaction {...};
};
```

Figure 6: Excerpt of Interface PersistentObject.

The method saveMe() is responsible for writing the object state, represented by its attribute value as present in volatile storage, into a persistent storage location. This involves creating a transaction, calling saveMeT() (discussed below) and closing the transaction as well as error handling. In case of an error—for instance if the data repository is out of memory—this method has different options to react, depending of the boolean value undo_when_failed. If it is false then the method just raises an exception, the object state is not saved, and the object keeps its local values. As a result, the object states in volatile and in persistent memory are different, which clearly is an unsatisfactory situation, to be handled by the application. Otherwise, the method reloads the original data from persistent storage by calling loadMeT() that undoes attribute changes before raising an exception. As a result, object values in volatile and in persistent storage are identical.

The persistent storage service can be used to store multiple objects in one transaction. In this case, saveMeT() is called. Rather than opening a transaction on its own (like saveMe() does), it writes the object values by using a transaction, passed to it. This renders feasible deep persistence of an object, i.e., the storing of multiple objects related to each other by object references in a single transaction. On the technical level, saveMeT() analyzes the object to get hold of its structure, i.e., its attributes and their respective data types. This meta information is used to call a write method for each attribute to save; the write methods are implemented within transactions objects, discussed below.

The methods loadMe() and loadMeT() mirror saveMe() and saveMeT(), to restore object attributes; loadMe() creates an transaction and calls the method loadMeT() to read the attribute values of a persistent object. After the read methods have been performed, it closes the transaction. Like saveMeT(), the method loadMeT() also analyzes the object structure. This information is used to call read methods within the transaction object to retrieve the attribute values from persistent storage. Just as saveMeT() is used to store multiple objects in a single transaction, loadMeT() allows to load a set of

objects from persistent storage in a single transaction.

3.4 Interface Database

The database interface (figure 7) describes the structure and behavior of data repositories. It also specifies how data repositories can be used to store persistent objects. The interface definition is generic in the sense that different database management systems as well as other data repositories like file systems can be used to store objects persistently. For each type of data repository, an implementation of the database interface has to be provided. For example, there is an implementation of the database interface for the Oracle 8 [9] relational database system, and there are implementations for IBM DB2 UDB [3] and for POSIX file systems. As is explained in detail below, this feature allows to distribute the set of persistent objects of an application among multiple data repositories that may even have different types.

```
interface Database {
  Transaction allocateTransaction() raises(Failed);
  void        freeTransaction(in Transaction t)
                raises(Failed);

  string getNewObjectmarker() raises(Failed);

  PersistentObject bindMarker(in string marker)
                     raises(Failed);
};
```

Figure 7: Excerpt of Interface Database.

Each data repository available for storing persistent objects is represented by an object of the class Database. This class is able to create transactions by the allocateTransaction() method. Notice that the differences in how to create a transaction in different systems is hidden by the database interface – it is provided by the respective implementations of that interface, for example, for the Oracle relational database system. Transactions can be considered interfaces between the object and the data repository. After object values have been written (or read) the transaction is released by the freeTransaction() method. We mention that for each new object, a unique object identifier has to be fetched by calling the getNewObjectmarker() method for the respective object.

3.5 Interface Transaction

The Transaction interface (figure 8) provides methods for storing attribute values and object information. The interface is generic, in the sense that its methods are independent from the internal data representations. We experimented with the following representations: one relation per class, object flattening, and binary streaming.

In the first alternative, all objects of a given class are stored in a specific database structure, e.g., a relational table. In this case, attributes are mapped to rows of matching type. As a result, data manipulation languages (such as SQL in the relational database context) can be used to access object attributes. Data access is typically fast, since the

query optimization facilities of the database system can be used and, furthermore, a single relation has to be touched to access any given object.

The object flattening technique uses one generic structure within the data repository to represent objects of all classes. Staying with the relational database system data repository, each attribute type is represented by one dedicated relation. This approach is very flexible, since class modifications have no impact on the database structure. SQL can be used to store and retrieve data, typically done by joining relations. Data access is typically slower than in the first representation technique, since join operations can be rather costly in database systems.

In binary streaming, object information is stored as a binary stream, for example generated by a memory dump or—in case the Java programming language is used—by the serialization interface, provided by Java. Implementations of the transaction interface using binary streaming are rather simple, but they do not allow data access by data manipulation languages. In addition, selection of result values has to be done in the application, rather than in the database, which may also incur additional overhead.

```
interface Transaction {
  void start() raises(Failed);
  void commit() raises(Failed);
  void abort() raises(Failed);

  void pushWorkingObjectMarker(in string marker)
         raises(Failed);
  void popWorkingObjectMarker();

  void writeObject(in string type,
                   in string server,
                   in string host) raises(Failed);
  void readObject(out string type,
                  out string server,
                  out string host) raises(Failed);

  void writeAttrLong(in string name,
                     in long value) raises(Failed);
  void writeAttrFloat(in string name,
                      in float value) raises(Failed);
  [...]

  long  readAttrLong(in string name) raises(Failed);
  float readAttrFloat(in string name) raises(Failed);

  [...]

  void deleteObject() raises(Failed);
};
```

Figure 8: Excerpt of Interface Transaction.

The transaction interface is explained as follows. A transaction has to be initialized with the start() method. The next step is the pushWorkingObjectMarker() method that tells the transaction which object data will be accessed, i.e., stored or loaded. The transaction implements a stack data structure: The object on top of the stack is the working object. Using the popWorkingObjectMarker() method pops the object from the stack. This stack functionality is useful if multiple objects have to stored, as discussed above.

The writeObject() and readObject() methods are used to access the object meta

information like its class, the name of the server process, and the machine name of the object. The object attributes are accessed by the

writeAttr$< type >$() and readAttr$< type >$()

methods, for instance writeAttrlong() and readAttrlong(). In particular, for each simple data type (e.g., byte, short, long, float, double, string), a method is provided. The transaction is closed by calling a commit() or abort() method that makes the changes persistent or rolls them back. If the object has to be destroyed, the object values are deleted from the data storage by the deleteObject() method.

To conclude the discussion of the interface transaction, we mention that transactions represented by objects of the class Transaction are used as interface between the persistent storage and application objects that want to store their attribute values persistently. These transactions are responsible for the storing (and loading) of attribute values to (from) data repositories. Hence, transaction objects do not implement a specific transaction service, for instance the Transaction Service Specification of the OMG, which is used to synchronize resource allocation between CORBA objects.

3.6 Sample Persistent Object Interface

This section explains the functionality and use of the persistent storage service by an example. For the time being, we will remain at the IDL level when discussing how to use the persistent storage service; the way this service works is explained without specific implementation details.

```
#include "CosPersistency.idl"

module Example {
  interface Person : CosPersistency::PersistentObject {
    string get_name();
    void   set_name(in string value)
             raises(CosPersistency::Failed);

    short  get_yearOfBirth();
    short  set_yearOfBirth(in short value)
             raises(CosPersistency::Failed);

    void   delete() raises(CosPersistency::Failed);
  };

  interface PersonFactory {
    Person create(in string name, in short  yearOfBirth)
             raises(CosPersistency::Failed);
  };
};
```

Figure 9: IDL Specification of Sample Persistent Object.

The sample application objects are of class Person; objects of that class have two attributes: name and yearOfBirth. These attributes are accessed by four methods: Each attribute has a method to set and to get the respective value. Since objects of class Person should be persistent, they inherit from the class PersistentObject. The interface definition of the Person interface is shown in figure 9.

The usage of the persistent storage service is very easy at the interface definition level: Persistent objects just inherit form the class PersistentObject. Methods that change the object‘s state just need the ability to raise the CosPersistency::Failed exception if they do not handle storage failures autonomously.

The life cycle of a Person object is investigated, from creation to deletion: When creating an object of the Person class, the create method of the PersonFactory object creates the object in volatile storage, and it initializes according to its definition. In the next step, a new object identifier is fetched by the getNewObjectmarker() method of the database object that represents the data repository in which the object should be stored. The identifier is assigned it to the persistent object. Then the database object allocates a transaction by allocateTransaction() and calls the saveMeT() method of the object. The creation is finished by freeing the transaction and – if a CORBA middleware is used – registering the object to the object request broker.

The saveMeT() method of a Person object analyzes the object itself or uses the objects meta information and generates in our example two write operations: writeAttrString() and writeAttrShort(). Now the object is ready and we can distribute the object reference to other objects and servers. If requests to the object appear, the methods of the object are executed in normal manner, i.e., independently from the persistent storage service. While the get_name() and the get_yearOfBirth() methods are not affected by the persistent storage service, they just return the attribute values from volatile memory, the set_name() and set_birthdate() methods have to call the saveMe() method after the changes before exiting the method. If the requested object is not in volatile storage, however, loading the object from persistent storage is required. Now assume that the server process holding the application object goes down, involving the loss of volatile object data. In this case, the object has to be reloaded into the volatile memory before the processing of the object request. This happens as described in Section 3.2; it is totally transparent for the object user and object implementor. The delete() method of the Person class unregisters the object from the object request broker, creates an transaction, calls the deleteObject() method of the transaction and finally removes the object from the volatile memory space after a successful closing of the transaction.

3.7 Design Goals Revisited

In this section, we match the conceptual design against the design goals. Where appropriate, examples are used to clarify the argumentation.

- *Ease-of-Use:* Persistent storage services based on the propsed framework are easy to use, since no additional classes have to be specified and implemented to store objects persistently. In particular, to store objects of a class C persistently, it suffices that C inherits from the PersistentObject class. By calling a single method saveMe(), the object state, i.e., the values of its attributes, are stored persistently. If the attributes of the object are of standard data types, no additional coding is required. Due to its extensibility, attributes of complex data types are also stored, provided the transaction interface includes respective methods. We sketch below how new methods can be defined and implemented to store complex values. In any case, no additional classes have to be developed to store

objects persistently, which reduces the amount of persistence support and simplifies maintainability of the the application code.

- *Class Evolution:* The proposed service supports modifications of classes. Changing the class structure will not affect the existence of an object, assuming a given class is not deleted. While changes of object behavior does not effect the persistent storage service, changes of object attributes may have.

  When an new attribute is inserted in a class, a default value for the new attribute has to be defined, and all objects of that class are provided with that value. In general, there are a number of strategies following the definition of a new attribute in a class. First, the default value for all objects with respect to the new attribute can be defined at build time. This means that the data stored persistently has to be changed after the object definition is changed. This alternative is not very elegant, because the modification can be done only by direct database calls, which require an understanding of the storage policy of the persistent storage service for the application programmer. The second alternative is to define a defaultValue() method in the PersistentObject class, which is responsible to set the value. In this case, the loadMe() method executes the defaultValue() method each time it finds an attribute that is not represented in the database. The method defaultValue() gets information about the missing attribute name and returns the correct value for this attribute.

  Deleting attributes may result in deletion of stored values of objects regarding this attribute. However, if storage space is not an issue then values of deleted attributes can remain in the persistent storage. Loading object states will simply not access these values, which is correct, since the corresponding attributes have been deleted. If, however, storage space matters then obsolete values can be deleted physically.

- *Data Integrity:* The methods to save and load objects are executed transactionally. To store a complete object persistently, methods to store each of its attribute values are executed. The transactional behavior of the persistent storage service guarantees that either all write operations are successful or none is performed. Depending on the data repository used, the transactional behavior comes for free (e.g., relational database system), or it has to be implemented in the transaction interface. Assuming a relational database system as a data repository, the all write operations that belong to a given object are executed as a single transaction. If the transaction completes successfully, i.e., if it executes a commit operation, then all attribute values of that object are stored correctly in the database. If the data cannot be stored successfully, the database management system executes an abort operation, and the values are rolled back, implementing the all-or-nothing semantics of transactions. If a non-transactional data repository is used (e.g., a POSIX file system), the respective functionality has to be implemented in the transaction class.

  Transactional features can be implemented by explicitly writing before images of all attributes that are changed, followed by writing the current attribute values. If not all current values can be written successfully, the before images have

to be restored for all attribute values that were already modified. We believe that for most applications, database systems are the appropriate choice as data repositories.

- *Data Repositories, Distribution and Heterogeneity:*
  Different data repositories can be used to store objects within the proposed interface definition, for instance, relational database systems, object-oriented database systems, and POSIX file systems. We appreciated the possibility to use different data repositories during a project, which in early stages suffered from run-time issues due to poor performance of the underlying database system, which caused performance problems on the application level. We decided to change the implementation to use a POSIX file system. The database and the transaction classes were re-implemented within a short period of time, and the system worked with increased performance without any changes in the application classes. This of course is due to the object design and the separation of object interfaces and their implementations. However, this practical experience demonstrated the usefulness of the object-oriented paradigm in this context.

  There is another reason why the possibility to use different data repositories without affecting the application classes is attractive, due to migration aspects. When an organization decides to use new data repositories, complex migration issues might occur. Using the proposed service, the specific differences of the data repositories involved are hidden by the transaction interface. For instance, substituting Oracle 8 [9] with IBM DB2 UDB [3] (or vice versa) can be done without any changes in the application code.

- *Data Representations:* As discussed above in some detail, the interface Transaction can be implemented to support different data representations. It is important to notice that these alternative storage representations are available within a single interface, which allows to change the storage structures without effecting the application. As mentioned above, this feature of the proposed service is another aspect of its extensibility. As discussed above, this feature can be used in software development projects to use simple and inefficient data representations in prototyping phases and switching to more elaborate data representations in later project phases, without any changes to the application objects.

- *Storage Policy and Granularity:* As shown in the example in Section 3, calls of saveMe() can be used within method implementations. For instance, in the end of a setName() method of an object of class person a call to saveMe() stores the effects of setName() in persistent storage. If and when setName() terminates successfully, i.e., and there are no exceptions, we can be sure that the call to saveMe() was successful, and the correct object value is stored persistently.

  The conceptual design introduced above provides the flexibility to store individual attributes of objects, i.e., just parts of objects. This feature can increase the overall system performance considerably, since individual attributes of objects that did not change since the object was written previously do not need to be updated. Exploiting this feature, however, requires to record which attributes

have changed since the previous saveMe() method was executed. For example, consider an image or video stream object with a text field, representing textual remarks. If the binary attribute remains unchanged for long periods of time, while the description attribute does change frequently then considerable gains in efficiency can be achieved, since only the textual attributes have to be written to persistent storage, while the large binary attribute does not. Hence, depending on the application, a flexible persistent storage service with the possibility to store different granularities (i.e., sets of attributes versus the complete object) of objects can lead to considerable performance gains. This shows another facet of the extensibility of the proposed service.

4 Implementation Aspects

As a prove of concept, the persistence framework has been used to implement a persistence service. The service was used as a fundamental component of an object-oriented workflow management system [10, 22]. This section discusses some implementation aspects.

The implementation is based on Java and OrbixWeb, Iona's CORBA object request broker implementation [8]. The Oracle relational database system and POSIX file systems are used as data repositories. There are three main components of the implementation, discussed below: the loader, the persistent data store and the component that is responsible for the collection of the application object‘s structure. We remark that the CORBA standard version 2.4 has a standardized interface for loaders that are represented by a specific interface, called ServantActivator; it is defined in the Portable Object Adaptor interface [16] of the standard. It is important to remark that this interface is compatible in its functionality with the OrbixWeb LoaderClass interface, so that our sample implementation can be ported to any CORBA 2.4 compliant implementation.

As a data representation we chose object flattening technique and an Oracle database as data repository. The connection between the persistent storage service and the database uses a Java Database Connectivity (JDBC) interface. The JDBC interface allows an access to the database by using SQL statements that are represented by Java objects. Each SQL statement may be parameterized; it is compiled at creation time of its representation object. This feature reduces run-time overhead, since the database system can use the pre-compiled code to execute SQL queries. As a result, the Transaction interface is implemented by a set of precompiled and parameterized SQL statements. In particular, for each method like writeObject(), readObject(), writeAttrLong(), readAttrLong() etc. there is at least one statement. When a method is invoked (e.g., readAttrLong() called by loadMe()) this method sets up the parameters of the precompiled SQL statement and executes it. Finally the method extracts the output parameter from the result set of the database query and transfers it to the object to be restored.

Using binary streaming makes the implementation of the Transaction interface much easier. We used the java.util.Hashtable class as a basic structure. All object information is stored in an object of that class. The java.util.Hashtable allows to store

(key, value)-pairs, like (attribute name, attribute value). Consequently, methods like readAttrLong() or writeAttrLong() need only to access the hash table. At the beginning of each transaction, a hash table has to be read from the file system using the java.lang.serializable interface. At the end of write transactions, the hash table has to be rewritten.

To save or restore attribute values, the methods saveMeT() and loadMeT() have to analyze the object and determine its structure, i.e., its attribute names, data types and values that have to be stored, or loaded. The sample implementation uses the java.lang.reflect package to collect this information during run-time. This package provides interfaces that allow the collection of object meta information at run-time. In particular, each Java object provides the method getClass(), which returns a meta information object, which in turn contains the method getDeclaredFields() to gather attribute names, attribute types, and attribute values. We use this information to call the transaction methods to store or restore object attributes. This approach provides a high degree of flexibility, as far as class evolution is concerned. For example, adding new attributes to a class can be done without modifications of the persistent storage service, since the structure of objects are determined by the Java package at run-time.

CORBA as well as Java use interfaces to specify objects. An interface is a description of methods for an object without the actual method implementations. In the sample implementation, CORBA interfaces are mapped to Java interfaces that are called operation interfaces. The exact mapping algorithm is described in [17]. As a result, the implementation for a CORBA interface is described in a Java implementation class that inherits form the Java operation interface, allowing multiple inheritance of CORBA interfaces, even though Java does not support multiple inheritance of classes. Object-oriented programming languages with full support of multiple inheritance like C++ are able to translate CORBA interfaces directly into native classes [18].

5 Related Work

This section overviews approaches and systems related to object persistence. Generic object-oriented persistence frameworks and Java-based approaches are discussed in turn. To demarcate our work from related approaches, we discuss why inter-object transactional properties are not in the scope of a persistent storage service and, hence, not in the scope of a persistence framework. We remark that OMG proposes distinct services for transactional behavior and persistence.

5.1 Generic Object-Oriented Persistence Frameworks

Much effort has been spent in recent years to provide persistence to object-oriented applications: The Object Management Group (OMG) aims at providing persistence for CORBA applications by specifying Persistent State Service [5], superseding a Persistent Object Service [14] that people found too complex and cumbersome to implement and use.

The OMG Persistent Object Service provides a high degree of flexibility in managing persistent objects. As a result, its design is rather complicated, which leads to

complex implementation and usage of the service. To store objects of a given class persistently, a new DataObject interface has to be implemented that takes care of reading and writing objects. To create objects of a particular class, a DataObjectFactory class has to be specified and implemented for that class. To allow for a transparent creation of persistent objects, a GenericFactory has to be defined that is aware of the DataObjectFactory of the objects to create. To summarize, the Persistency Service as specified by OMG generates considerable overhead, because it requires to implement multiple classes for each application class, whose objects should be persistent. We mention that the Persistent Object Service is superseded by the Persistent State Service, discussed next.

The OMG Persistent State Service ensures durability of object data by introducing a new layer above the CORBA Interface Definition Language (CORBA IDL) layer, called Persistent State Service Definition Language, or PSSDL layer. In the PSSDL layer, the persistence behavior of data types is described. PSSDL is a superset of IDL, i.e., PSSDL extends IDL with semantics for persistence. Once the persistence behavior of a given class is defined in PSSDL, that code is transformed by a compiler into IDL and programming language code, implementing persistence for objects of the class.

The main functionality of the PSSDL layer is a storage type specification and a storage home specification. The storage type specification describes the data types of objects, stored in a database. This specifications are translated by the compiler into database queries that create required database structures and functionality to access stored data. For example, if relational databases are used, create table commands and select statements are generated by the compiler. The storage home specification describes the storage relevant functions, for instance create or find_by_key_name. During run-time, access to persistent objects is done by creating sessions in which the persistent objects are incarnated ("loaded") by using key search mechanisms. At each instant, there may be several sessions, in each of which exactly one persistent object may be incarnated. As a result, a given persistent object may have several incarnations that reside in different server processes. Consequently, a persistent object may have several different object identifiers, which clearly amounts to a problem in object-oriented environments, where each object should have exactly one identifier during its life time.

Introducing a new layer dedicated to persistence results in a high complexity of the development system for the application developer. On the one hand, she has to learn a new specification language (PSSDL), on the other hand she has to deal with an extra layer of abstraction. As a result, application developers have to cope with three layers of abstraction: PSSDL, IDL and implementation code.

5.2 Java Persistence Frameworks

With the success of the Java programming language, a variety approaches to add persistence to Java have been proposed. Although our approach is not restricted to Java, these are important contributions to be discussed.

With Enterprise JavaBeans (EJB), Sun Microsystems proposes a component model for distributed object-oriented business applications, developed in Java [20]. In the EJB approach, business objects are designed, implemented, and deployed as software components, known as enterprise Java beans. EJB are implemented by application servers.

Making use of the standardization, applications are expected to run without changes in application servers of different vendors. As far as persistence is concerned, an EJB application consists of two kinds of beans: entity beans and session beans. While entity beans are used to represent persistent objects like customers, contracts, and orders, session beans are used to represent transient objects. Session beans are typically used to model activities or sets of related activities. For instance, a session bean TravelAgent may invoke operations like reservations, orders, and customer billings. Application servers provide persistence for entity beans. Depending on the implementation, tools for mapping object attributes into relational tables are provided.

In the EJB approach, there are a number of additional classes to define and implement for each persistent application class. We sketch the usage of persistence in the EJB approach by objects of a class ship [11]. A remote interface Ship is required that represents the business methods for objects of that class. To access specific objects, a primary key class ShipPK is required. In general, for each application-specific class, a respective primary key has has to be specified and implemented. A class ShipHome acts like a factory for ship beans, i.e., it is responsible for creating and deleting ship objects. Finally, a shipBean class is the implementation class that implements load, store, remove, among other methods for the persistent storage of ship beans. Without going into the details of programming in the EJB component model, this discussion shows that a variety of additional classes are required to store objects persistently. In contrast, the approach presented in this paper requires no additional classes. To some extent, the complexity of persistence in EJB is introduced by the much broader scope of the EJB component model. As a consequence, using enterprise Java beans just for the persistent storage of application object seems inadequate. EJB do make sense, however, if additional functionality provided by application servers is required by the application.

In the context of enterprise Java beans, the following remarks on object representations are in order: While mapping tools provide different representations of object values in relational or object-oriented databases, our approach is more flexible, since individual attributes can be retrieved from persistent storage, whereas in the EJB approach the complete object is written to or read from persistent storage. In our approach, users control if and when a persistent object is written to or read from persistent storage. For example, a number of consecutive operations on a large persistent object can be performed in memory, and only after the last operation is completed, the object value is stored persistently. These operations can be executed in transactional fashion, to make sure intermediate results do not corrupt persistent data. In the EJB approach, however, the decision on when to store persistent objects is taken by the application server, which may lead to performance issues when there are multiple operations on a given large object that are executed sequentially.

Gemstone/J is a platform for web-based business applications. These applications are implemented by Java servelets that access databases, legacy systems or other enterprise applications by the Java Database Connectivity (JDBC) interface. For performance reasons, Gemstone/J allows the distribution of servelets on several Java virtual machines. Gemstone/J provides services for distribution aspects, for instance load balancing, JDBC connection pooling, caching, and transaction monitoring [6]. Gemstone/J aims at supporting business to business cooperation by providing access to the applications of one business partner via web-based front-ends. In contrast, CORBA

aims at supporting business to business cooperation by the integration of information systems of both business partners. The integration is done by wrapping business logic into business objects.

Java Blend is a tool for the development of database applications in Java. The tool and its API encapsulate relational database queries and update statements. As a result, the application programmer deals with objects and methods rather than with data and queries. In particular, Java Blend allows the definition of classes, whose objects represent rows in a database relation. The application programmer may use transactions and OQL (Object Query Language) queries to access objects. The result of a query is an object or a set of objects that can be manipulated by methods, defined for that object. Technically, a pre-compiler analyses the application code and inserts Java code for classes and statements that are needed to access data. One problem of Java Blend is the fact that each row of a database table is represented by several transient objects. However, a row in a database relation does not naturally correspond to a real-world object, which is disadvantageous for the implementation of objects in distributed environments.

The main focus of Java Blend is the implementation of database applications in Java. Hence, the persistence aspects of Java Blend focus on data retrieval and not on the the persistent storage of business objects, which is needed in business to business scenarios.

5.3 Non-Object-Oriented Approaches

Atkinson introduces the principles of orthogonal persistence that are tailored to the persistent application system developement using procedural programming languages. The three principles are [1]:

1. *Persistence independence*: The form of a program is independent from the longevity of the data that it manipulates. Programs look the same whether they manipulate short-term or long-term data.

2. *Data type orthogonality*: All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.

3. *Persistence identification*: The choice of how to identify and provide persistent objects is orthogonal to the universe of dicourse of the system. The mechanism for identifying persistent objects is not related to the type system.

These principles make sense in procedural application programming; however, they can not be fully adopted for distributed object-oriented environments. While persistence independence makes sense in object-oriented environments, the other principles conflict either with the object-oriented paradigm or with the distribution aspect. Data type orthogonality makes no sense in object-oriented environments, because object-orientation makes use of inheritance, meaning that object properties (e.g., persistence) are defined by the class, the object inherits from. The third principle, persistence identification, is easy to implement in non distributed environments, for example by identification by reachability. This means, an object is persistent if it is reachable form a

persistent root. Since object references are non-local in general, it is quite expensive and hard to implement identification by reachability in distributed environments. As a result, persistent objects have to be identified using other techniques. For instance, it makes sense to identify persistent distributed objects by their type, e.g, by inheritance of the PersistentObject class.

To summarize, Atkinson proposes a set of important properties for programming environments supporting persistence, mainly concerned with procedural and non-distributed approaches. Hence, this work is not directly applicable to our approach. However, the principle of persistence independence is satisfied by our approach.

The Grasshopper orthogonal persistent operating system is an attempt to resolve the strict separation between main memory and backing storage by the introduction of an additional abstraction layer [4]. The idea of Grasshopper is to provide an operating system that automatically manages the persistence of all programs executed in the system. From a program‘s point of view, only one (partitioned) memory space exist, programs need not differentiate between volatile and persistent memory. This approach makes in fact all executed programs persistent without involving the programmer. The Grasshopper operating system presents a promising approach to a generic persistent storage solution. However, some non-trivial problems need to be solved: recovery and efficiency. Since processes are inherently persistent, there are no transient processes, resulting in performance issues, since all processes are stored persistently.

6 Discussion

In the final section we like to deliver our thoughts towards integrating the persistence framework with the OMG Transaction Service as well as on optimizing the persistence framework. Finally we present our conclusions and plans for future work.

6.1 Integration with OMG Transaction Service

Transactional properties involving multiple objects is provided by dedicated transaction services, for instance the Transaction Service specified by OMG [15]. Since object-oriented applications typically require persistence as well as transactional behavior, it is an important requirement that a transaction service can make use of a persistent storage service. In this section we sketch how the OMG Transaction Service and persistent storage services based on the proposed persistence framework can be integrated.

Assume an object that has to be persistent and that has to provide transactional capabilities. That object inherits both from the PersistentObject interface (of the persistent storage service) and from the TransactionalObject and Resource Interfaces, provided by the Transaction Service [15]. Essential parts of the Transaction Service are shown in figure 10.

An object that inherits from the TransactionalObject interface has the ability to provide transactional behavior; the application programmer can make use of that service without additional programming. The Resource interface deals with the objects' storage policy, i.e., this interface defines under which conditions object attributes are stored or recovered to provide transactional behavior.

```
interface Resource {
  Vote prepare() raises(HeuristicMixed,
                        HeuristicHazard);
  void rollback() raises(HeuristicCommit,
                         HeuristicMixed,
                         HeuristicHazard);
  void commit() raises(NotPrepared,
                       HeuristicRollback,
                       HeuristicMixed,
                       HeuristicHazard);
  void commit_one_phase() raises(HeuristicHazard);
  void forget();
};

interface TransactionalObject {};
```

Figure 10: Excerpt of IDL for OMG Transaction Service.

To implement a persistent and transactional object, the methods specified in the Resource interface have to be implemented. The method rollback() can be implemented easily with the persistent storage service by calling the loadMe() method. That method restores persistently and consistent attribute values of an object in order to implement the rollback. As shown in figure 10, the Resource interface supports a one-phase-commit. This means that all objects involved in a transaction will receive commit requests. As opposed to the 2PC protocol, there is no voting on the success of the transaction. However, since the Resource interface supports the 1PC, we sketch this approach. The commit_one_phase() method has to be overloaded that way, that the saveMe() method is executed to store the object values persistently. We remark that the method forget() needs no special coding.

The implementation of the two phase commit protocol [2] is more advanced, thanks to the more complicated communication model. The object has to open a database transaction (using the CosPersistency::Transaction interface), store the object attribute values as usual and finally send a prepare-to-commit to the database (in case this functionality is supported by the data store). If the used data store does not support prepare-to-commit, the persistent storage service has to simulate the two phase commit protocol, for example by storing the object values in a persistent buffer space during the preparation phase until the commit() method is invoked. Finally the commit() method forwards the commit to the data store, in which case the object values in the data store are overwritten by the buffer data. In case the transaction has to be aborted, the buffer data can be neglected, and the original object values as stored in the data store remain untouched.

6.2 Complex Data Types

We mention that complex CORBA data types and non-CORBA data types are not directly supported by the proposed framework. However, due to its extensibility, the proposed service specification can be extended to handle arbitrary data types, including complex CORBA data types and non-CORBA data types. The design decision not to support a variety of data types *a priori* is due to the object paradigm, which stipulates that complex data should be described by objects rather than by complex data types.

The proposed specification can be extended to support complex data types. The modifications to the interface definitions of the relevant classes and the methods to be added are sketched. Instead of implementing relationships by a list of pointers to the related objects, a relationship should be represented by objects of a relationship service, for instance the one specified by OMG. Extending the service to handle complex CORBA data types is sketched below.

In scenarios where only a few different complex CORBA data types are required the most efficient way to extend the transaction interface of the service with writeAttr<Type>() methods of the new data types. In the example sketched above, the type sequence<Object> is required to handle relationships. For example, by adding a method writeAttrSequenceObject(), sequences of objects can be stored within a single method invocation. To implement this method, either the sequence of objects is stored in a relational table (in case data is stored in a relational database system), or the implementation makes use of sequence storing mechanisms that object-oriented database systems provide. If support for arbitrary CORBA data types is needed, a more advanced implementation have to be provided. In this case, the transaction interface is extended with a writeAttrAny() method that at runtime analyzes the data structures and stores them. Assuming a serialized storage policy, this method is quite easy to implement.

6.3 Efficiency and Optimization Aspects

As stated above, the efficiency of the proposed persistent storage service is not discussed in detail, because it depends to a large extent on the data representation. However there are three general optimization aspects that are feasible within the persistent storage service specification, motivating its extensibility from an optimization point of view:

1. *Minimize data to store*: The persistent storage service should transfer only changed attributes to the database. If an object consists of large attributes it may be useful to store this attributes only on request of the programmer. An alternative could be to backup object attributes and store only the attributes that have changed. For large object attributes it would be useful to store a checksum instead of the whole attribute. The checksum can be used to detect changes to the attribute value. If the stored checksum equals the new checksum then the attribute value was not changed and, consequently, does not need to be written.

2. *Minimize number of database transactions*: If several objects have to be stored at the same time, they should be stored in one database transaction instead of using several database transactions. Especially in combination with a transaction service it is a valid approach to store all objects involved in a high-level inter-object transaction using only one (low-level) database transaction.

3. *Avoid synchronized communication points*: The communication between an object and the database object should use as few method calls as possible. This means for example instead of using several writeAttr<Type>() methods in the transaction interface, the transaction interface should be enhanced with a method

that is able to deal with a sequence of (attribute, attribute name) pairs. This approach would improve the efficiency of the system because the object data can be stored using one message. This means only one synchronization point instead of several synchronization point like before.

Again, these optimization aspects can be implemented without any changes to the interface definition of the persistent storage service, showing its extensibility in the context of implementation and optimization aspects.

6.4 Final Remarks

This paper presents a persistence framework for object-oriented middleware. While its conceptual design is independent from a specific middleware, a sample implementation uses the CORBA product OrbixWeb. The service is easy to use, since it suffices to inherit from the PersistentObject class in order to make objects of a given class persistent. As explained above, the persistent storage service provides a high degree of extensibility and flexibility, since it supports different data repositories and data representations, even within a given application. By providing the possibility to store specific attributes of objects, i.e., the attributes whose values have changed recently, performance gains can be achieved.

Future work will center around using the service in real-world scenarios. We believe that for high-performance applications, a re-implementation of the service in C or C++ will be required. Another issue to be addressed in the future is the development of a configuration tool for the service that can make use of the extensibility of the service by allowing to tailor it to match the requirements of a broad range of applications.

Finally we remark that the persistent storage service was developed in the context of a research project on flexible workflow management [22], where it was used to store workflow schema objects, workflow instance objects, and application, i.e., business objects. Hence, it has undergone extensive checking, since thousands of persistent objects are required for complex workflow scenarios, all of which were handled by the proposed persistence framework and its sample implementation.

References

[1] Atkinson, M.P., Morrison, R.: *Orthogonal Persistent Object Systems.* VLDB Journal 4, 3 (1995) pp 319-401

[2] Bernstein, P.A., Hadzilacos, V., Goodman, N.: *Concurrence Control and Recovery in Database Systems.* Reading: Addison-Wesley 1987

[3] Chamberlin, D.: *Using the New DB2: IBM's Object-Relational Database System* San Mateo: Morgan Kaufmann 1996

[4] Dearle, A., Vaughan, F., di Bonda, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J.: Grasshopper: An orthogonally persistent operating system. Computing Systems 7(3), pp 289–312, 1994

[5] Fujitsu Limited, Inprise Copr., Iona Technologies Inc., Objectivity Inc., Oracle Corporation, Secant Technologies Inc., Sun Microsystems Inc., TIBCO Software Inc.: *Persistent State Service 2.0 Joint Revised Submission.* OMG Document orbos/99-07-07. OMG 1999

[6] GemStone: *GemStone/J 4.0 The Adaptable J2EE^{TM} Platfrom* (available at http://www.gemstone.com/products/j/index.html)

[7] Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques.* San Mateo: Morgan Kaufmann 1993

[8] Iona Technologies: *OrbixWeb Programming Guide.* Dublin: Iona Technologies 1996

[9] Koch, G., Loney, K.: *Oracle8: The Complete Reference.*

[10] Kuropka, D.: *Specification and Implementation of a CORBA-Based Persistent Workflow Engine.* (in German) Diplomhausarbeit. Institute for Information Systems, Universität Münster 1998

[11] Monson-Haefel, R.: *Enterprise JavaBeans^{TM}* . Sebastopol: O'Reilly 1999

[12] Mowbray, T.J., Zahavi, R.: *The Essential CORBA: Systems Integration using Distributed Objects.* New York: Wiley 1995

[13] Object Management Group: *CORBAServices: Common Object Services Specification.* OMG Document 98-12-09 1998

[14] OMG: *CORBA Services – Persistent Object Service.* OMG Document formal/97-12-12 (Chapter 5). OMG 1997

[15] OMG: *CORBA Services – Transaction Service.* OMG Document formal/00-06-28. OMG 2000

[16] OMG: *CORBA/IIOP 2.4 Specification. Chapter 11: Portable Object Adaptor.* OMG Document formal/01-02-01. OMG 2001

[17] OMG: *IDL to Java Language Mapping Specification.* OMG Document formal/99-07-53, OMG: 1999

[18] OMG: *C++ Language Mapping Specification.* OMG Document formal/99-07-41. OMG: 1999

[19] Orfali, R., Harkey, D.: *Client/Server Programming with Java and CORBA.* Second Edition. New York: Wiley 1998

[20] Sun Microsystems (Matena, V., Hapner, M.): *Enterprise JavaBeans^{TM} Specification, v1.1.* Palo Alto: Sun Microsystems 1999

[21] Sun Microsystems: *Java Blend Application Programming Guide.* (available at http://www.sun.com/software/javablend/index.html) Palo Alto: Sun Microsystems 1999

[22] Vossen, G., Weske, M.: *The WASA2 Object-Oriented Workflow Management System.* Proc. 1999 ACM SIGMOD Conference on Management of Data, pp 587–589. New York: ACM 1999