Design and Implementation of a Replicated Persistent Object State Storage

J.E. ARMENDARIZ1, J.J. ASTRAIN1, A. CORDOBA1, J. VILLADANGOS2
1Dpt. Matemática e Informática, 2Dpt. Automática y Computación,
Universidad Pública de Navarra
Campus Arrosadía, 31006 Pamplona (Spain)

ABSTRACT
The aim of this work is to provide persistent object storage support over an RDBMS for an existing object replicated environment called COPLA. Specifically, an object repository is viewed by an application as unique, consistent and fully available. It provides persistent object state storage under a replicated environment, granting a transactional behaviour. Object state access is performed by means of specific methods or by queries performed by final application users. Object recovery facilities on faulty nodes and the evolution of applications developed in this architecture are also provided.

An object definition language and an object query language, based on the ODMG standard, have been designed so as to define applications and query objects inside COPLA respectively, and how objects are translated into the entity-relation model.

Keywords: Persistence, Replication, RDBMS, ODMG, ODL, OQL, CORBA

1. INTRODUCTION

Many enterprises have delegations in different cities and even countries. Obviously, these delegations need to access and store persistently data that is shared among them. Whenever a delegation failure occurs, a whole system failure may take place. In order to prevent this situation, it is necessary to replicate or distribute the information between different sites. We have adopted the former, denoting data replicated at a given site as a repository. This must provide a unique view and a uniform access to data throughout the whole system to applications that handle the information, both following the object-oriented paradigm.

Replication mainly consists of copy maintenance and its client availability. Motivations that have lead to the usage of replication techniques are the following: performance enhancement, availability increase, and, fault tolerance.

Geographically distributed replication implies that the server response time is decreased. Data, shared among a large client community, should not be included in a single server, mainly because it will act as the system “bottleneck” both in response time and in throughput (measured in processed requests per second). It is often desirable to distribute data copies among different servers, and configure the system in different little client sets that access the nearest replicated data server measured using network metrics terms, e.g. network congestion and so on.

Two of the main requisites for data replication are replication transparency and data consistency. Replication must be performed transparently for final users. From their point of view, information is viewed as an individual logic data unit, and they refer to these units when they request operations over this data. Clients are not aware about the number of physical copies neither the physical data copy where they are performing operations. Data consistency in a replicated environment consists of guaranteeing that applications will not see different results when they access the same logic data unit. These logic data units must be stored in a non-volatile device, which is what stands for persistent data storage, so that a server failure will not imply information loss in the replicated architecture.

The aim of this work is to provide persistent object storage support for an existing object replicated environment called COPLA (Common Object Programmer Library Access) [1]. The architecture proposed supplies to application developers a unique layer that includes and simplifies shared object access. An object repository is viewed by an application as unique, consistent and fully available. This paper novelty related to previous works, up to our knowledge, is that it provides persistent object state storage service, called Uniform Data Store (UDS), under a replicated environment, granting a transactional behaviour and the chance to access object state by means of specific methods or by queries performed by final application users. Other original features are: object recovery possibility on nodes where a failure has been produced, ensuring throughout the whole system a consistent object access without any notice to the rest of the components of the system; and, the evolution of applications developed in the COPLA architecture (application scalability).
Service main goal is to provide a unique software layer that isolates the rest of the architecture components of the persistent object storage details. We use a RDBMS as the persistent storage device, due to its technology maturity and its popularity either in the system administrator and final user environments, besides it allows to easily migrate previously data stored in the RDBMS to o-oriented client applications.

It is important to note that we propose an architecture that provides object replication support while the persistence mechanism follows the entity-relation model [2,3]. Thus, this mismatch between both is solved using an object definition language following the ODMG standard called Globdata Object Definition Language (GODL). It establishes how classes, attributes and relationships must be defined, as well as their relational mapping over the RDBMS. Moreover, this implies a new consideration, due to the fact that the object state is stored in an RDBMS. Afterwards, this process must be reversed, they must be translated to a given programming language class, so as to be properly treated by an application.

COPLA provides an object transactional access and object lookup mechanisms of them by means of object-oriented queries. Therefore an object query language, again based in the ODMG standard must be defined called Globdata Object Query Language (GOQL), or by way of the appropriate method invocation of the corresponding class. In order to supply a consistent object state access, the service must provide the following features to COPLA: persistent storage of metadata concerning to consistency maintenance; transparent update information propagation; and, node recovery of areas that could have failed, taking into account different consistency protocols requirements.

The rest of the paper is organized as follows: Section 2 introduces related works about replication and persistence; Section 3 explains the system architecture where the service has been implemented; Section 4 and Section 5 describes the object definition language and object query language used for applications; Section 6 depicts the persistent object storage service; and, finally, conclusions end the paper.

2. RELATED WORKS

Data updates propagation techniques in replicated environments may vary from asynchronous techniques, where actualizations are propagated when it is suitable or periodically, to the totally synchronous, which update order of data replicas is total [4]. There exists different works showing different approaches to data replication. Thus, in [5] a replication system is proposed, completely Java based, by means of RMI. This system provides communication among a group of servers, all of them provide Java objects replication and it also provides modifications on the client side, which makes it possible the recovery against object register failures. The Aroma system [6], provides also object replication using RMI, but the main difference with the latter is that it uses interception mechanisms to provide a transparent object replication, utilizing object grouping and offering a totally secure and ordered multicast. There exists more generic solutions, i.e. it does not depend on a given programming language, like the CORBA for transparent object replication by means of grouping techniques [7]. These works lack of a persistent object storage service for objects created, they need object instantiation during runtime execution and remain in the main memory while the application is running. Once they are not used anymore, they disappear and there is no way to recover them again.

Design and implementation of persistent storage systems have been treated in the literature [8,9]. They deal with object state storage in a non-volatile media such as a hard disk or a database management system, either relational or object oriented. We have based our system implementation in some of them. There are implementations developed for concurrent environments [10,11]. Others use CORBA so as to guarantee object persistence [12]. Persistence based on mobile agents has been treated in [13], while the persistence of Java objects for an enterprise intranet or specific web applications has been developed in [14].

3. REPLICATED ARCHITECTURE

The result of this work is the implementation of a software layer that isolates the rest of COPLA components from persistent implementation in the RDBMS. It provides additional features that guarantees update replication of data from one node to the rest of nodes of the architecture [26]. COPLA architecture manages system failures and application performance. Both aspects are closely tight, because implementing more robust techniques leads to a system performance decrease. This system supports repositories failures, which will be able to recover. Obviously, the failure of a node will affect data availability. An overview of the system is given in Figure 1.

![Figure 1: System architecture.](image)

Several nodes, called COPLA managers, divide the system in several areas. They serve to applications defined in their respective areas. Applications only access its “closest” manager, that is, the manager responsible of that area. All applications have facilities in order to localize other managers, which are used when its local manager fails. A manager is in charge of one UDS. Thus, it provides a view of a unique centralized object repository in the whole system. Clients interact using a set of classes grouped in libraries, called COPLA libraries. They are in charge of
representing the object state on the client side, that is, each application corresponds to an object repository or GODL schema. Each COPLA library interacts directly with the COPLA manager belonging to its area, which provides UDS access in order to properly work.

Each UDS contains all object repositories (modules) defined in the system. What it is not guaranteed, because it depends on the particular consistency protocol used [1], is that objects contained in a UDS are the newest version of them stored in COPLA. A transaction started by a client application is applied over the UDS of its respective local COPLA manager, using data stored in that UDS. At a given time, there may only be one consistency protocol running in all managers of the system so as to guarantee data consistency. It performs coordination tasks and sends to the applications the latest version of objects requested by them.

If a manager fails, the system continues its activity until it is feasible to continue working, assuring always data consistency and integrity. Applications have a registry where they have stored all the managers of the system, so when its manager fails, they will be able to continue working connecting to a new manager.

COPLA's goal is to provide access to a unique logical object repository, which in fact contains several physical replicas (see Figure 2). Applications use for their definition GODL. This language characterizes an object repository, denoted as module or schema, on which classes, their attributes and relationships are defined. In this definition, a module is the same as an object repository, which is composed by classes, where each one is composed by attributes, either literals or objects, and the relationships with other classes. These objects may be accessed by way of object-oriented queries, by GOQL. Applications developed in the COPLA architecture follow two properties. The first one is, high degree of "locality", it means that areas covered by each COPLA manager writes over set of objects whose intersection is the empty set. This means that write conflicts on COPLA architecture are practically non-existent among COPLA managers. Conflicts, if they exist, will mainly occur inside the manager where the application was defined. The second property is that some areas need only the object version to maintain consistency. Thus, it is not necessary to transfer completely objects, it is only needed to know if an object they are trying to access is still valid.

4. GLOBDATA OBJECT DEFINITION LANGUAGE

GODL is a subset of ODL. Its main restriction is that it does not allow nested collections, since this is not a primary target of our work. In this section we describe the GODL grammar, whose main elements are modules, classes, attributes and relationships. A module component allows programmers to define a GODL application, or schema. Classes are defined inside a module, such classes define a set of attributes and relationships, and it may extend another previously defined class. Attributes are like properties built from simple types (like integers), strings or instances of other classes. It also considers multivalued types such as collections of literals (or objects), which consist of bags, lists, sets and arrays. Relationships are associations that consist of a reference to a collection of them. An example of a module (or an application) definition is given in Figure 3.

```java
module example {
    class address {
        /* Attributes */
        attribute short zip_code;
        attribute date moving_date;
    };
    class person {
        attribute string<100> name;
        attribute long dni;
        attribute enum state (Navarra, Madrid, Valencia);
        attribute address address;
        ...
    };
    class student extends person {
        attribute set<short> grades;
        ...
        /* Relationships */
        relationship set <course> takes
        inverse course:is_taken_by;
        ...
    };
    class course {
        ...
        relationship set <student> is_taken_by
        inverse student:takes;
        ...
    }
}
```

**Figure 3: Definition of a GODL schema.**

Since objects defined in an application must be persistently stored in an RDBMS, we have to define a translation schema from GODL to RDBMS that will be afterwards described. It also must adapt GODL to Java, since clients handle Java applications. We have developed a GODL compiler, that is Java based, which has been implemented utilizing JLex and CUP tools as the lexical analyzer and the parser generator respectively. It translates a given module to its equivalent entity-relation model and its equivalent Java structure for the client side, which we will refer as proxies. An object identifier (GOID) is used to univocally identify an object inside the architecture.

The procedure followed to persistently store a module into an RDBMS is to map each class into a new table, this table will contain the following fields: the GOID and all the univalued literal attributes defined for this class. A new table is created for each object attributes, the table contains at least two fields (in case of an univalued, bag or set object attribute), each one of them refers to the field of its
is well formed when each projection is defined in the respective class, i.e. the owner GOID and the aggregated GOID. Besides, it may contain an additional field to maintain the order when a list attribute is given, or, as many fields as the array dimension when we deal with array attributes. The same process is followed when multivalued literal are given, the first field refers to the owner GOID, the second one stores the given value, and there will be additional fields when list or array attributes are defined.

Relationships defined in the given module are mapped into new tables, following a schema similar to multivalued attributes, but in case of a relationship that includes a list, an extra field is added to reflect the order in the given relationship. Moreover, two extra fields are added when we deal with list-list relationships.

These collections are handled by appropriate triggers and SQL stored procedures. Additional tables are required as metadata: the aggregation table, that manages the aggregation between objects; the inheritance table, that contains all the hierarchy trees; the attributes table, every attribute defined in this application is included in this table, reporting about its belonging class, its type, its cardinality and the array dimension; the relationships table, that describes the classes related, the name of the relationship and its inverse and the cardinality of both; and, consistency protocols metadata, being as many tables as different protocols have been implemented [1]. An outline of our mapping proposal is given in Figure 4.

5. GLOBDATA OBJECT QUERY LANGUAGE

This language allows us to retrieve collections of objects or literals that verify certain conditions. GOQL is based on the OQL standard, but differs in some features. The main difference is that we do not allow the definition of new data types in the query language. This restriction has been imposed in order to obtain a consistent object repository due to the fact that we are inside a replicated architecture. But some useful characteristics have been added to improve its functionality.

It retrieves collections composed by literals or objects and allows to navigate through attributes and relationships defined for a given class by means of path expressions. It contains two mandatory clauses SELECT and FROM, and two optional clauses WHERE and ORDER BY.

The main components used in GOQL to construct a query are described here. These expressions are: class, is considered as a collection of non-repeated instances of this class; variable, is an alias which refers to a collection of objects, it is used to iterate over each item inside the collection; GOID; parameter, is an alias that represents an object or a literal, it will be replaced with the assigned value once the query was compiled; and, path expression, is an expression that allows to travel through objects and reach the required data. A path expression is composed by a source and one or more projections. A path expression is well formed when each projection is defined in the previous partial object retrieval and the full path expression does not return nested collections.

Our proposal, followed in the GOQL compiler design and implementation, is to construct a compiler that is able to translate object oriented queries into relational queries, following the object-to-relational mapping proposal. The translation process is helped by means of the meta-information stored in the database. During the definition of GOQL we look for a single SQL query as the result of a translation of a GOQL query. Then, we could not add all the OQL functionality to GOQL. There were some operators and even concepts in OQL that could not be easily implemented over a relational database, because they may involve several SQL sentences and some post-processing of the results, which we have not considered. We have optimized the translation for PostgreSQL.

CREATE TABLE ADDRESS(GOID VARCHAR REFERENCES AGGREG(GOID) ON UPDATE CASCADE ON DELETE CASCADE, STREET VARCHAR(100), ZIP_CODE INT2, MOVING_DATE DATE);
CREATE TABLE PERSON(GOID VARCHAR REFERENCES AGGREG(GOID) ON UPDATE CASCADE ON DELETE CASCADE, NAME VARCHAR(100), DNI INT8, STATE VARCHAR CONSTRAINT STATE CHECK(STATE = 'Navarra' OR STATE = 'Madrid' OR STATE = 'Valencia'));
CREATE TABLE STUDENT(GOID VARCHAR REFERENCES AGGREG(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));
CREATE TABLE PERSON_O_ADDRESS(PERSON VARCHAR REFERENCES PERSON(GOID) ON UPDATE CASCADE ON DELETE CASCADE, ADDRESS VARCHAR REFERENCES PERSON(GOID) ON UPDATE CASCADE ON DELETE CASCADE); CREATE TABLE COURSE(GOID VARCHAR REFERENCES AGGREG(GOID) ON UPDATE CASCADE ON DELETE CASCADE, NAME VARCHAR(100), DNI INT8, STATE VARCHAR CONSTRAINT STATE CHECK(STATE = 'Navarra' OR STATE = 'Madrid' OR STATE = 'Valencia'));
CREATE TABLE STUDENT_TAKES(STUDENT_TAKES VARCHAR REFERENCES STUDENT(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));
CREATE TABLE STUDENT_L_GRADES(STUDENT VARCHAR REFERENCES STUDENT(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));
CREATE TABLE STUDENT_L_GRADES(STUDENT VARCHAR REFERENCES STUDENT(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));
CREATE TABLE PERSON_O_ADDRESS(PERSON VARCHAR REFERENCES PERSON(GOID) ON UPDATE CASCADE ON DELETE CASCADE, ADDRESS VARCHAR REFERENCES PERSON(GOID) ON UPDATE CASCADE ON DELETE CASCADE); CREATE TABLE COURSE(GOID VARCHAR REFERENCES AGGREG(GOID) ON UPDATE CASCADE ON DELETE CASCADE, NAME VARCHAR(100), DNI INT8, STATE VARCHAR CONSTRAINT STATE CHECK(STATE = 'Navarra' OR STATE = 'Madrid' OR STATE = 'Valencia'));
CREATE TABLE STUDENT_TAKES(STUDENT_TAKES VARCHAR REFERENCES STUDENT(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));
CREATE TABLE STUDENT_L_GRADES(STUDENT VARCHAR REFERENCES STUDENT(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));
CREATE TABLE STUDENT_L_GRADES(STUDENT VARCHAR REFERENCES STUDENT(GOID) ON UPDATE CASCADE ON DELETE CASCADE, GRADES INT2, CARD_POS INT4, CHECK (CARD_POS >= 0));

Figure 4: Overview of our mapping proposal.

We have set up a cache and defined the grammar and syntax to perform parametric queries. The compiler has a fixed size with a least recently used policy, which stores the SQL translation (among other information) of a GOQL query performed by the user. If the feature of parametric queries is added to this compiler then it is only needed to compile once and, from that moment on, it is only necessary to substitute and check that the appropriate parameters are inserted.

In the following, we will give an outline of the query process translation. The SELECT clause translation consists in replacing, if it is an object, the variable that refers to the GOID field of the respective table name, otherwise if it is a literal, it is substituted by the respective field of the table in the database. The FROM clause translation is formed by the
cartesian product of individual extensions of each one of the variables involved. The translation of these extensions is performed by means of JOINs of the involved classes with a CROSS JOIN of the class extension. The WHERE clause results in a translation process similar to the previous one if it includes a path expression. Otherwise, the respective fields of the tables are used in order to verify the condition. The ORDER BY clause follows the same translation pattern.

Continuing with the application defined in Figure 3, we will give some GOQL examples: “SELECT s.dni FROM student AS s;” is translated to “SELECT s_.dni AS TARGET FROM (student JOIN person USING (GOID)) AS s_”.

6. UNIFORM DATA STORE

UDS [26] is a service that stores persistently the objects state in an RDBMS by way of transactional accesses. The UDS will be used by the respective consistency protocol. The objects states are stored following the mapping proposal depicted in Section 4. It also keeps all the metadata needed by the UDS in order to properly locate an object, either by means of a direct method invocation or by way of a GOQL query. It maintains meta-information needed by consistency protocols in order to properly work. This information mainly consists of the object version, the object owner, etc. It is important to note that we are talking about an information replication service. Thus, the system providing persistence and the consistency protocols need to communicate each other. The intercommunication mechanism used is CORBA, which is the middleware used in the rest of the COPLA architecture. Thus, several IDL interfaces have been defined that establish how COPLA communicates with the UDS for information exchange in order to perform several tasks. The UDS must implement all the previous interfaces in Java [26], by remote operation invocation between them and the response reception by means of its serialization.

The main aim of this service is to provide a persistent object state storage of instances defined in an application inside the COPLA architecture, guaranteeing a transactional behaviour and the possibility to access them either by specific method invocation or by GOQL queries. The results obtained by the implementation are introduced in this part. Transactions are created so that applications may have object repository access and consistency protocols may access their metadata and propagate their updates. UDS creates transactions in the RDBMS, which guarantee concurrent access to an object repository and to metadata for the consistency protocols.

Protocols, depending on the information they have available, may decide whether a transaction ends properly (commit) or decide to abort (rollback), when a final user decides to commit (or rollback) its updates. This is due to the fact that conflicts may appear about the consistency of objects accessed or updated by other current transactions in the system. The information used for this task by a consistency protocol consists of the enumeration of objects read and written in transactions opened throughout all UDS nodes. Once the transaction has been committed, this service provides facilities to propagate transparently all the updates performed to the rest of nodes.

The UDS interacts with an RDBMS by way of the JDBC API with the appropriate PostgreSQL driver. In order to supply a more efficient object repository access, it has been established: a pool of JDBC connections; a data structure storing all the information dealing with replication (it stores information about objects read and written along the current transaction); as well as the SQL sentences that have modified the object repository during the current transaction.

Another task of this service is to properly locate attributes and relationships defined for each class. Therefore, taking into account the inheritance of that class, it must build the appropriate SQL sentences for their location in the RDBMS. This information interchange (creation, update and querying) between the UDS and the consistency protocols is done by means of the definition of the respective IDL interfaces, which define the way this process is performed. User may perform queries about the state of objects created inside a GODL module, by means of GOQL. Query evaluation always returns a collection. The query compilation process requires two stages in COPLA. Depending on the consistency protocol used at that time, it may be necessary to update the object repository of the node where the query is being compiled. This first phase consists of the translation from GOQL to SQL, besides the UDS sends the classes and GOIDs involved in the query, allowing to update classes and objects so that the user has a consistent view of the object repository.

The persistent service has been implemented over a replicated architecture. It must provide mechanisms so that COPLA may propagate the updates performed in one object repository of a node to the rest of nodes. The UDS is responsible for defining the format on which updates are propagated from one node to another. An object inside COPLA is completely defined by all its attributes and relationships with other classes, throughout its hierarchy tree. If COPLA decides to move an object from one node to another, a recursive translation must be performed. This process must be repeated as many times as the depth of its hierarchical tree. Thus, sometimes it is more interesting to move the whole repository rather than locating all the information with successive SQL queries to the database and building IDL structures, that leads to a decrease of the system performance. The structure used to move one object from one node to another is through a PackedObject. It is an IDL structure defined in the COPLA architecture. It contains all the univalued attributes and relationships throughout its hierarchy tree. This is used to move the information to its respective proxy.

We have only considered fail-stop errors in the COPLA architecture. At this moment, the consistency protocol will consult its corresponding meta-information to determine which objects are present in the UDS recently joined. If there are no differences among versions stored, it means that it contains the latest versions of all objects and it is not necessary to perform any recovery process. Otherwise, it
must perform an update operation. The consistency protocols may adopt the solution of moving the whole object repository. On the other hand, if objects updated are a few, it may only move objects that have been created or modified while the current node has been off-line. Another important task of the UDS is the collection management. Methods defined for their manipulation, are the ones specified in the ODMG. This implies the construction of a SQL sentence for the required elements inside the RDBMS. This query can be generated by: a GOQL query performed by a client and translated to SQL by the GOQL compiler; or, accessing a method on the client side, on which the UDS must build the equivalent SQL sentence to retrieve the elements requested by the final user. All objects read and written are stored in its respective read and write set of the current transaction.

7. CONCLUSIONS

This work introduces a persistent object storage service to be used by an existing object replication architecture called COPLA. This architecture consists of several areas extended over a wide area network. We have defined an object definition language, called GOQL, that defines a COPLA application, or module, that contains classes with its hierarchy, attributes and relationships. The objects of this application are going to be persistently stored in an RDBMS, and at the same time represented on the client side by their respective proxies. A translation procedure has been introduced in order to solve the mismatch between the object oriented model and the entity relation model. Users may retrieve objects persistently stored by direct invocation of the corresponding methods or by means of an object query language, that we have called GOQL, whose grammar and behaviour was introduced in this paper.

The UDS is a unique layer that isolates the rest of the COPLA architecture of the persistence implementation details over an RDBMS. It provides the following features: a transactional access to an object repository inside an RDBMS, and it properly locates all information regarding to a given object repository, either by GOQL queries or methods; it supplies additional methods to update propagations to different consistency protocols; it has defined a set of meta-information that deals directly with the metadata needed by the consistency protocols in order to properly work; and, mechanisms for recovery tasks.

8. REFERENCES


