Persistent Object Synchronization with Active Relational Databases

Fábio A. M. Porto\textsuperscript{1}, Sérgio R. Carvalho\textsuperscript{2}, Maurício J. Vianna e Silva\textsuperscript{3}, Rubens N. Melo\textsuperscript{4}
Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225, Rio de Janeiro, RJ, 22453-900, Brazil
{porto\textsuperscript{1}, sergio\textsuperscript{2}, rubens\textsuperscript{4}}@inf.puc-rio.br, mjvs@domain.com.br

Abstract

One of the most common client/server architectures in enterprise systems today is the combination of object-oriented applications with active relational database systems. With this combination, developers have to overcome a difficult problem: the impedance mismatch between object orientation and the relational model. To date, there are several incomplete approaches for describing the integration of static and dynamic object aspects and active relational databases. An important issue missing from these approaches is the state synchronization between server tuples and client-cached objects. In a previous paper we proposed a technique for mapping the dynamic behavior of objects into active relational databases, using database triggers and stored-procedures. This paper extends our previous one with an architecture based on a replication strategy that maintains server tuples and client-cached objects synchronized with respect to state. This architecture automatically updates client-cached object versions when their corresponding server database tuples are updated.

Keywords: object orientation, active relational databases, state synchronization, data replication, object cache, stored procedures

1 Introduction

For more than a decade, business critical applications have been representing their information assets in relational database management systems (RDBMSs) [Duhl96]. More recently, advances in network technology, distributed programming and software architectures propose the organization of business applications in different tiers spread through various computers [Buschman+96, Aarsten+96, Hirschfeld96].

The combination of object-oriented applications with relational database systems within a client-server architecture is probably one of the most common choices for enterprise systems today [Delis+98]. Unfortunately, this combination still many issues to overcome: the impedance between the Object Oriented (OO) model and the Relational model (RM) requires different approaches to deal with structural and behavioral clashes [Keller+96].

On the structural side, for example, object attributes may be stored in different database tables; also, object relationships such as inheritance have no counterpart in the relational world. On the
behavioral side, which we address in this paper, state changes in application objects must be reflected on their persistent versions, and vice-versa.

Pattern languages have been proposed to bridge the existing gap between the two technologies [Brown+96a, Keller+96, Silva+97]. In [Porto+98], we discussed this issue with respect to object behavior. Our proposal allows the representation of object life cycles within active RDBMSs, implementing object behavior via database triggers and stored procedures. By encapsulating dynamic behavior into the RDBMS we simplified the client side of the application and reused all rule enforcement mechanisms provided by commercial database systems.

However, the action part of server rules can, in many cases, change the state of stored objects. The problem of representing these changes in the client side presents itself. The objective of this work, which extends the previous one, is to propose an architecture for the automatic update of client object versions from alterations performed on the database version of the object. We therefore continue to address behavioral OO/RM clashes, neglecting to consider structural mismatches.

In this work, client-server applications with persistent objects stored into RDBMSs can be perceived as instances of an environment with replicated object versions. Once a set of persistent objects is read into the client station, at least two versions of the same object exist: a persistent version, stored into the RDBMS, and an application version to be accessed by the user in the client machine.

The remainder of this paper is organized as follows. Section 2 describes the object synchronization problem in more detail. Section 3 presents the proposed object synchronization architecture. Section 4 shows the scenarios where data are updated and synchronized. Section 5 exemplifies the use of the architecture with a simple application scenario. Section 6 compares our solution with related work. Section 7 presents conclusions and future research.

2 The Synchronization Problem

Consider an application implemented with an OO programming language and an active RDBMS. The application runs on a PC type desktop and the RDBMS on a server machine. Clients start the application and request data from the RDBMS, thus loading the application business objects. Once data gets loaded into the client application, the user deals with it as an independent OO application. The relational implementation becomes entirely transparent [Keller+96].

Application objects become versions of the persistent relational data, cached at the client machine, or at the application layer environment. The reasons for creating such data versions are initially to offer OO semantics for the application data stored in a RDBMS, and secondly to increase overall performance by splitting the application between various collaborating environments.

Considering updating applications, object versions in each client and within a (possibly distributed) server can be modified by user actions in some client’s environment, and by active behavior in the RDBMS, via pre-defined triggers and stored procedures [Widom+96, Porto+98]. Thus, in the presence of either application-side or database-side updates, object versions cached in client environments become out of date, so-called stale cached data [Franklin+97].

This happens for three main reasons:

In order to increase overall throughput, persistent application classes implement an optimistic concurrency control protocol [Gray+93], so data read from the RDBMS is not locked while being processed by the client.

User updates take place offline, over local data versions. This strategy greatly increases client execution performance and provides an improvement for overall RDBMS data concurrent access. Updates go to the RDBMSs only at the end of the transaction, during the commit processing.
Server logic, implemented as stored procedures and triggers, may update persistent data that might have been cached and asynchronously updated within the client environment.

With increasing numbers of application clients and application update rates, the lack of synchronization within object versions may turn out to be critical. Client data lag far behind their persistent versions. In such a scenario, a great number of transactions may have to be completely re-submitted.

Considering the above, update-intensive applications can benefit from a proposal that addresses the state synchronization between client-side and database-side objects, thus aiding in the resolution of the OO/RM impedance mismatch. In the next section we present such an architecture.

3 The Object Synchronization Architecture

Figure 1 shows the classes and relationships we propose. The ConcreteApplication, ConcretePersistentApplication and Transaction classes are structured in ways very similar to patterns proposed in [Keller+96, Keller98, Silva+97]. The service classes ApplicationLog, CopyManagerClient and CopyManagerServer provide our proposed functionality.

3.1 Abstract Application

The AbstractApplication (AA) class generalizes the manipulation of application objects with persistent behavior. Each domain specific class modeling objects with persistent behavior extends this AA class. These ConcreteApplication (CA) classes model objects in application domains. Persistent read and write operations over CA class objects are passed to the corresponding ConcretePersistentApplication object, described below.

The AbstractPersistentApplication (APA) class provides a common persistent behavior for domain specific classes, including the special messages implementing the synchronization mechanisms between client objects and database tables. ConcretePersistentApplication (CPA) classes extend APA, modeling the persistent behavior of domain objects.

Each CPA is a Singleton [Gamma+95] responsible for the mappings between corresponding object views and relational tables. All communications involving object versions and RDBMS tables pass through one such CPA object, which acts as a broker. The kind of structural
mappings that may be executed by a CPA class are those presented in [Keller+96]. They represent the main OO structural constructs, such as class, inheritance hierarchy, and association.

The Transaction class singleton controls client application transactions. The responsibilities of this singleton are: to generate transactions identifiers, to register new client transactions with the CMC object, to inform the CMC object of the success or failure of the corresponding transactions, and to apply updates over RDBMS data when a commit operation successfully concludes a client transaction.

The DynamicTransaction (DT) class extends the Transaction class with special behavior needed to process client transactions that execute stored-procedures, which in this work implement object behavior in the RDBMS. The DT object queries and updates data in the auxiliary tables used to inform the CMS object of updates executed by server procedures over persistent data.

The ApplicationLog (AL) class also models a Singleton. It registers the update operations executed by CA objects using a write-ahead policy [Gray+93]. Its object collaborates with the Transaction object during the commit processing, providing all the operations executed within the transaction’s boundaries.

The CopyManagerClient (CMC) class models a Singleton responsible for the communication among all CPA objects and the Transaction object with the CopyManagerServer object. It provides communication transparency between clients and the CopyManagerServer object. All operations over cached data will be asynchronously informed by the CMC object in each client environment to the CopyManagerServer object, during transaction execution.

The CopyManagerServer (CMS) class models a Singleton multi-threaded server in a 3-tier environment, serving all RDBMS clients. It registers transactions running on client machines together with a list of table names corresponding to the objects loaded by the transactions. This data structure is used by the CMS as a directory for sending synchronization messages, which inform registered clients with cached versions of persistent objects, of updates committed by the RDBMS server procedures or by concurrent client applications.

We may consider the architecture as split in three tiers. Composing the client environment we have the following classes: AA, CA, APA, CPA, AL, Transaction, DynamicTransaction and CMC. In the middle tier runs the CMS server object and the RDBMS composes the third tier.

In the next section we describe a few scenarios illustrating the synchronized behavior we obtain with the architecture above.

4 Update Scenarios

The architecture presented in section 3 aims to reduce the time lag between a confirmed database update and the moment in which cached versions of the corresponding data have their attribute values synchronized. The database modifications are applied either by applications running on client machines and executing SQL statements through database connections, or by server procedures running on the database server machine.

This section integrates our synchronization architecture with the scenarios in which database data are modified. Our main concern is to present the scenario where server procedures update object data stored in the RDBMS. This is the case when we implement object behavior through database stored-procedures and triggers. We also discuss the modifications in database data executed by client applications. This scenario is divided in two parts: the process of loading and

\footnote{It is important to note that not all object views might be translated into tables. In special, non updateable views [Silberschatz+96, Elmasri+94] like the ones containing aggregated values do not map into tables modeled over the analytical data.}
updating objects and the commit process. These scenarios are complemented with a fourth one that identifies and communicates registered clients of stale data.

In all four scenarios the main participants are the CMC and the CMS objects. They provide data structures and operations for supporting the synchronization process. We initiate the presentation of the update scenarios describing the responsibilities of these components.

The CMS object records, for each transaction in a client, a list of tables that have been accessed, and most importantly, have been updated during transaction execution.

The application successfully terminates a transaction by issuing the commit operation on the Transaction object. This event causes the staleness of cached versions being manipulated by other client environments. In order to synchronize data, the Transaction object informs the CMC object of the transaction’s commit. The CMC object then passes the information to the CMS object through the InformCommit message. It’s a function of the CMS object to inform registered clients that their data versions are out-of-date.

Once the CMC object, controlling a registered client transaction, receives a message about the existence of new versions of stored data, it notifies interested ConcretePersistentApplication objects. When all the clients have acknowledged the message, the CMS object destroys the object corresponding to the finished transaction.

4.1 Client side updates

Figure 2 and Figure 3 show scenarios where a ConcreteApplication object is updated. Object modifications are imposed by transaction operations. Transactions are initiated by calling the BeginTransaction( ) operation on the Transaction object. Each update operation over persistent ConcreteApplication objects informs the transaction object controlling its progress.

After having been associated to a transaction object, the ConcreteApplication object uses it in its communication with the CMC object. All information interchange references the transaction object, so that in the event of a commit, it becomes possible to identify which tables had their states changed during the transaction’s processing.

![Figure 2 Object Synchronization Architecture Scenario – load and update Objects](image)

Execution on the client side progresses with almost complete independence from the RDBMS side. Once an object is required, i.e. via an user interface request, the CPA object requests the proper tuples from the RDBMS and composes the object’s view corresponding to the required ConcreteApplication object, for instantiation purposes. ConcreteApplication objects have a timestamp (ts) attribute. During instantiation, they receive the value of the oldest timestamp among its component tuples. This attribute will be used, during the RDBMS transaction commit, to validate the consistency of the cached version versus the RDBMS version.
An update in a ConcreteApplication object starts the synchronization process. This object informs its corresponding CPA object that an object has been updated. Considering the OO/RM mapping, the CPA object identifies the corresponding tables structurally associated with the updated object. Next, it informs the CMC object of the tables updated. Note that, if the object is composed by tuples in different relational tables, the CMC objects registers the update in all such tables. When the user decides to commit the transaction, it invokes the corresponding Transaction method. The commit process begins by identifying the update operations executed during the transaction, obtained by demanding the ApplicationLog object to provide the net effect of the operations executed during the period \cite{Widom+96}. The CPA translates operations executed over objects into relational counterparts over tables. Once the set of operations is formulated, the Transaction object attempts to execute them within a single RDBMS transaction. During database updates, the corresponding client objects are locked, guaranteeing a consistent synchronization of views.

The Transaction object waits for the RDBMS’s return from the commit operation. Following a successful return, it informs the CMC object of the transaction’s commit. This object in turn informs participant clients of new database versions of data, of which they have stale versions. This is done by first informing the CMS object of the committed transaction, via the InformCommit message, with parameters identifying the client and the transaction that committed. The CMS object then invokes all clients registered for the updated objects.

Data may also be changed by RDBMS’s server procedures. In particular, we propose the representation of client object behavior via triggers and stored procedures \cite{Porto+98}. These server procedures may update, as part of their code, data cached in client environments. Server procedures execute within the boundaries of a RDBMS transaction, controlled by the CPA object providing client object persistence. The CPA object uses a transaction object modeled by DynamicTransaction (DT) class to control transactions executing stored-procedures. The DT class extends the Transaction class overriding the operations for the creation and finalization of transactions.

To inform clients with cached object versions of updates executed by server code, we use three auxiliary tables: PersistentTable, UserTransactionTable and ProcedureUpdateTable. The PersistentTable is a meta-data table storing the table names and ids for the ones that have their states changed by server procedures. It serves two basic purposes: document tables representing
objects with behavior stored in the RDBMS, and provide consistency for data stored in the ProcedureUpdate table.

The UserTransactionTable is updated by the DynamicTransaction object during begin (insert) and end (delete) of transactions implementing object behavior. Its data represent the collection of transaction operations, through the association of the database user identification and the client’s transaction identification.

The ProcedureUpdateTable is updated by the server procedures implementing object behavior. Examples of possible modifications imposed by server procedures include deleting and inserting tuples in state tables, and the execution of pre- and post-conditions associated with a state transition. Each of these updated tables is registered in the auxiliary table together with the user-id and client transaction id.

Figure 4 presents the execution scenario for procedures and triggers implementing object behavior. The overridden methods of class DynamicTransaction are responsible for inserting and deleting user transaction information into the UserTransactionTable.

![Figure 4 Server procedure’s update scenario](image)

After receiving a successful return code from the executed procedure, the CPA object commits the RDBMS transaction. It then terminates the client transaction by issuing the overridden DynamicTransaction Commit operation. The overridden operation controls the execution of a sequence of operations aiming to identify the tables updated during server processing and allowing for the initiation of the synchronization process. It also destroys the transaction object.

The CMC operation InformCommitSP, invoked by the Commit operation, executes in two steps: firstly it queries the ProcedureUpdateTable, finding out tables updated by server procedures, as shown by the query bellow.

```
“Select pt.table_name, timestamp From PersistentTables pt, ProcedureUpdate pu
 Where pt.table_id = pu.table_id and pu.user = :username and
 pu.trans_id = :trans_id”
```

Secondly it deletes the corresponding tuples in the ProcedureUpdateTable, deleting the registration of updated tables. Having recovered the execution control, the Commit operation
deletes the tuple in the UserTransactionTable corresponding to the terminating transaction, deleting the transaction record.

Finally, it identifies the CPA objects associated with the updated tables and informs them of the RDBMS updates. To initiate the synchronization messages, the CMC object invokes the RegisterCommittedUpdate operation of the CMS object listing the tables that were updated during the execution of transactions.

### 4.3 Object level synchronization

The message sent by a CMC object to a CPA object informs that some data, corresponding to a view it controls, was changed. Considering that the granularity of the synchronization control exercised by CMC and CMS objects is a table, the CPA object is responsible for finding out if the change impacts some of the active objects under its scope. The CPA object invokes the GetObjects method of the corresponding ConcreteApplication object. The method returns a list containing the objects presently loaded at the client environment. Using its mapping rules, the CPA object queries the tables corresponding to its view. It uses the attributes composing the table primary key and the timestamp attribute value to identify the objects which need to be updated.

Basically, two sets of results are of interest. First, if no tuple is found for a primary key value, it means that the persistent version of the object has been deleted by some transaction. As a result, the version cached at the client must be destroyed. Second, if a database tuple exists for a primary key value but it presents a timestamp value greater than the one in cached object version, then the object persistent version has been updated. As a result, the object in cache must have its values updated.

Finally, with these queries the CPA object is able to identify the objects that had their states changed, lock them, update their versions and return a message to the user aborting the current transaction (see Figure 5).

---

**Figure 5 Object level synchronization**

**5 Related work**

The problem of integrating OO applications with relational database systems has been the subject of various studies in the literature. [Keller+96] presents a relational access layer. We use this access layer as the basis for our work, and introduce a caching policy. Our proposal extends Keller’s work with a treatment of the synchronization problem between client cached data and RDBMS server persistent data. The mapping rules implemented by the ConcretePersistentApplication class are those defined in [Keller+93]. In [Porto+98] we propose the modeling of object behavior through database stored-procedures and triggers. The patterns
presented in [Silva+97] extend those in [Keller+96]. In this work we use their StrongLayering pattern.

There is also a great deal of work associated with the replication strategy. For example, [Gonçalves+98] presents a pattern language for implementing architectures supporting object replication with different policies. It is a more general approach then that found in the Observer pattern [Gamma+95]. In our proposal, the CMS object is responsible for informing registered CMC clients of committed changes in the RDBMS. In a way, the CMC/CMS objects present a behavior similar to that found in the subject/observer metaphor.

Another area of related work investigates RDBMSs extended to manage client cached data. [Franklin+97] presents a taxonomy for algorithms that maintain the consistency of cached data. With respect to this work, our proposed concurrency control strategy relates to detection based protocols with validity checks deferred until the commit, and change notification hints sent after commit. We extend their approach dealing with object-cache consistency, taking into account object data that has been updated by stored procedures and triggers.

6 Conclusions and Future Work

One of the most common choices for software development these days combines three important paradigms: OO programming, active RDBMS and client-server architectures. The first two paradigms are not orthogonal, offering different modeling perspectives. The third paradigm often involves a distributed architecture on top of which one can distribute parts of the application. Putting all this together is not easy. Much work has been done in the OO/RM mapping. In this area one of the main concerns is the problem of synchronizing client-side object state changes and database-side table updates.

In a previous paper [Porto+98], we proposed the relational integration of application object behavior through database triggers and stored procedures. We used the active mechanisms of the RDBMS to execute object transitions, verify pre-conditions and execute pos-conditions. The execution of RDBMS server procedures, however, presents us the inverse problem: once the database code updates persistent data, the application objects become out of date.

In this paper we propose a solution to this synchronization problem, encompassing both the application-to-database solution described in [Porto+98] and a new solution to the inverse database-to-application problem.

The architecture we propose considers applications developed using an OO programming language. Persistent application objects are stored in relational database systems with the active capability of running server procedures. This architecture makes no further assumptions; the classes and relationships we propose can be implemented in any OO language, and the object-relational mappings may follow any proposed pattern language.

Persistent objects are created and processed in the client OO environment, as part of a client transaction running an optimistic concurrency control protocol. At the time a client transaction is confirmed, the application tries to store the object into the relational database. Processing in the client environment is almost completely independent from the RDBMS. The data requested from the database are loaded in the client environment without being locked by the server. Once the client environments run independently from the server, updates made by server procedures may interrupt an ongoing client transaction. Depending on the size of the client’s transaction, it may be very painful for the client to cancel all that has been done.

Our architecture aims to diminish the impact on client transactions by informing, as earlier as possible, of updates committed by other transactions that change the values of client objects. Our main concern is to inform clients of updates caused by server procedures and triggers during the processing of object state transitions.

Our architecture implements a combination of a replication strategy with client/server optimistic concurrency controls, and patterns for solving the OO/RM impedance mismatch. This
functionality may be summarized in three parts: the identification of updates over objects or tables, the registration of the updates and the broadcasting of the update message.

There are many opportunities for future work, examining alternative solutions to the problem. Our solution creates a RDBMS-like environment in the client. A possible alternative could be to use an OO DBMS in the client, simplifying part of the architecture. The control of updates by the CMS is done at table level; alternatively, we can control updates at the object granularity. Also, the communication between server RDBMSs and application components may be improved by eliminating the need to use auxiliary tables.

References


[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.


