MADIS: A Slim Middleware for Database Replication*

Luis Irún-Briz[1], Hendrik Decker[1],
Rubén de Juan-Marín[1], Francisco Castro-Company[1],
Jose E. Armendáriz-Iñigo[2], and Francesc D. Muñoz-Escoí[1]

[1] Instituto Tecnológico de Informática
Universidad Politécnica de Valencia – 46071 Valencia, Spain
{lirun,hendrik,rjuan,fcastro,fmunyoz}@iti.upv.es
[2] Dpto. de Matemática e Informática
Universidad Pública de Navarra – Campus Arrosadía s/n, 31006 Pamplona, Spain
enrique.armendariz@unavarra.es

Abstract. Data replication serves to improve the availability and performance of distributed systems. The price to be paid consists of costs caused by protocols by which a sufficient degree of consistency of replicated data is maintained. Different kinds of targeted applications require different kinds of replication protocols, each one requiring a different set of metadata. We discuss the middleware architecture used in the MADIS project for maintaining the consistency of replicated databases. Instead of reinventing wheels, MADIS makes use of basic resources provided by conventional database systems (e.g. triggers, views, etc) to achieve its purpose, to a large extent. So, the underlying databases can perform more efficiently many of the routines needed to support any consistency protocol, the implementation of which thus becomes much simpler and easier. MADIS enables the databases to simultaneously maintain different metadata needed for different replication protocols, so that the latter can be chosen, plugged in and exchanged *on the fly* as online-configurable modules, in order to fit the shifting needs of given applications best, at each moment.

1 Introduction

Providing distributed access to their databases is key for banks, warehouse chains and large enterprises with geographically widespread branches. Computer applications and services for such companies must cater for shared accesses to and transactions of local and global enterprise data, which may be distributed or replicated over several sites. With databases that are fully replicated in several nodes of the network, read accesses can be local if a ROWAA [1] policy is used, so that the availability of the data and the performance of the applications is improved. Employing replication techniques also benefits the fault-tolerance of the system, improving the ability of the database to be transparent with regard to local failures and to recover seamlessly.

However, replication has some important drawbacks. The system must introduce a potential overhead for maintaining the consistency of replicated data [2]. In addition, applications making use of replication necessitate additional pieces of software in order

* This work has been partially supported by the Spanish grant TIC2003-09420-C02-01.

© Springer-Verlag Berlin Heidelberg 2005

to manage the access to distributed resources, thus incrementing the complexity of their development.

In this paper, we describe a new middleware architecture, called MADIS[3], for supporting the distributed replication and hence the high availability, high perfomance and high fault tolerance of databases. It is designed as a two-layered architecture, with the aim to isolate the consistency manager (CM) as a module which is independent of any underlying DBMS particularities. Additionally, MADIS takes advantadge of existing database resources for efficiently achieving its tasks, so that the implementation and execution of protocols is not overburdened by the overhead usually entailed by the consistency management in replicated databases.

The upper layer consists of the middleware providing the protocol functionalities for replication and consistency management. The lower layer is an automatically producible extension of the original schema of a given database, using exclusively standard SQL features such as triggers and stored procedures, so as to provide to the upper layer the information needed to carry out its tasks efficiently. For instance, the set of records read, written, created or deleted in a transaction is automatically stored in a particular table of the extended database schema. The consistency protocol thus is able to retrieve that information, avoiding the use of otherwise necessary additional routines, which usually tend to be complex and error-prone. As a result, the mechanisms of the middleware to manage the collection, retrieval and removal of such meta-data have become much simpler, when compared to those needed in other middleware-based systems for replicated databases, such as COPLA [4]. Of course, the performance of a middleware-based replicated database will be worse than that of a core-based one, such as Postgres-R [5], but its advantage is to be independent of, and thus much more easily portable to other DBMSs. Moreover, the upper layer can be implemented in any programming language, since the support it needs is fully in the DBMS using SQL.

MADIS supports the pluggability of protocols, such that different kinds of protocols (ranging over various paradigms, from eager [6, 7] to lazy update propagation [8], from optimistic to pessimistic concurrency control, etc) can be modularly chosen, plugged in and exchanged, according to the shifting needs of given applications. An important feature is that protocol switching is seamless and fast, since it can be performed without having to recompute the required metadata for a newly plugged-in protocol. In general, the modularity of the system and the pluggability of protocols provide an unprecedented openness of the replication middleware.

The rest of the paper is structured as follows. Section 2 describes the structure and functionality of MADIS. Section 3 describes the schema modification that MADIS proposes to aid a local consistency manager (CM). Section 4 outlines an implementation of the CM, in the form of a standard JDBC driver. In section 5, a performance analysis studies the overhead of MADIS over an unmodified PostgreSQL schema. Sections 6 and 7 compare our approach with other systems and summarize the paper.

2 The MADIS Architecture

The architecture proposed by MADIS consists of two main layers, each one providing a number of functionalities. In essence, the lower of these layers consists of a modification in the schema of the underlying database repository, in order to provide and manage additional tables. We call these tables "report-tables". The upper layer intercepts requests from the user application, and makes use of the information stored in the report-tables to perform the consistency management.

The report-tables are automatically maintained in the lower layer. They contain information accounting the execution of various transactions in the local node. The modifications done in the report-tables are managed inside the same transactional context as the transaction which these modifications refer to. As the modification of the schema only uses SQL-99 features, a high degree of portability is ensured. A set of database procedures is also provided in the schema modification, in order to hide to the upper layer the details of the schema extension.

The upper layer of the MADIS architecture is positioned between the client applications and the database. It acts as a database mediator. Common accesses to the database as well as the commit/rollback requests are intercepted, allowing the consistency protocol to take part in the process. The consistency protocol can gain access to the incremented schema of the underlying database to obtain information about executing transactions, thus performing the actions needed to provide the required consistency guarantees. Finally, the consistency protocol can also manipulate the incremented schema, making use of the provided database procedures when needed.

The implementation of the upper layer (i.e. the Consistency Manager) can be done regardless of the underlying database. In this paper, we describe a Java implementation, designed to be used by the client applications as a common JDBC driver. The functionality this driver introduces regarding consistency control over a distributed database is provided in a transparent way to the user applications. The Consistency Manager is the core of MADIS. It manages database *connections* (which may include multiple sequential transactions, working in different JDBC consistency modes) and controls a set of database replicas. Moreover, it provides the plug-in for a *consistency protocol agent*, which can be chosen according to the requirements of the given application. The supported protocols share some common characteristics. All the communication performed between the networked databases is controlled by the local consistency manager.

3 Schema Modification

The lower layer of the MADIS architecture consists of a modification in the schema of the existing database. The process for distributing an existing centralized database starts with the execution of a program that performs a schema migration at each replicated node. This migration consists of the inclusion of tables, views, triggers and database procedures designed to maintain, automatically, a number of reports about the activity performed during the lifetime of a transaction. That way, the schema modification allows the database to automatically perform the collection and maintenance of transactions writesets, as well as the metadata pertaining to the different records in the database[1]. *Optionally*, it also collects and manages transaction readsets (possibly including the information read to perform queries). If this information is not generated, a

[1] As different metadata are needed by different consistency protocols, the extension caters for all of them.

consistency protocol requiring such information should perform some additional work from the upper layer.

The operations needed by the consistency protocols can be performed through a number of added database procedures, thus enabling an *ad-hoc* management (not always required) of the information automatically maintained in the database.

3.1 Modified and Added Tables

For each existing table T_i in the original schema, MADIS defines a number of modifications, relating field additions, view definitions, and others. Therefore, a new field is added for metadata purposes so as to identify a record on T_i, this new field is called *local_T_i_oid*. To this end, a field is added, defining a link to the metadata associated with each record in the table T_i

The attribute holds the local object identifier for the record. This identifier is local to a particular node in the system. Thus, it is possible for an object (identified by a *unique* global_oid) to have different *local_T_i_oid*'s within the system. A global_oid is required for the different nodes in the system, to agree in the identity of each record, regardless the local identification (sensible to local information).

In addition, MADIS creates for each table in the original schema (T_j) an extra table (named `MADIS_Meta_`T_j), containing the metadata needed for any protocol pluggable in the consistency manager. When a protocol is activated, MADIS executes a start-up process, to initialize each *"Meta"* table in the database. The primary key of the table consists of a unique object identifier. A typical *"Meta"* table is described as a tuple: **(local_oid *(pr.key)*, global_oid, version, transaction_id, timestamp)**.

The *MADIS_Meta_T_j* tables contain all the information needed by any replication protocol pluggable in the system. Hence, as all the fields are automatically maintained by the database manager, any of such protocols is suitable to be activated at will.

In addition to meta-tables, MADIS defines a table *MADIS_TrReport* containing a log including the activity of each transaction of the database. The table is as follows: (**trid, global_oid, field_id** *(optionally)*, **mode**). Where the primary key is composed by: (**trid, global_oid, field_id**). For each transaction, only one record per field-of-object is maintained, recording the access mode (**mode**) is recorded for each accessing transaction (**trid**), the global object identifier (**global_oid**) corresponding to the accessed record, and -maybe- the identifier for the accessed field within the record (**field_id**). In addition, once the transaction is terminated, the consistency manager eliminates from this table any record relating the concluded transaction. Note that several MVCC-based DBMSs (this is not the case of Postgress) do not use locks with record granularity, but locks that block access to entire pages or even tables. Such systems must use multiple "per transaction" temporary *TrReport* tables, including the transaction in the table name (i.e., these tables have a *<trid>_TrReport* name).

3.2 Triggers

As mentioned, MADIS introduces a set of new triggers in the database schema definition. These triggers can be classified in three main groups:

– *Writeset managers.* They are responsible for the collection of the information relating the objects written by the executing transactions.
– *Readset managers.* Collect the information related to the objects read by executing transactions. Their inclusion in the schema is optional, and when included, it is requested to be implemented by creating views.
– *Metadata automation.* These triggers are executed when the metadata stored in the MADIS extension tables must be updated. The collection and maintenance of such information is performed automatically by the triggers.

The writeset collection (WSC) is performed defining three triggers for each table T_i in the original schema. They insert in the `TrReport` table the information related to any write-access to the table performed by the executing transactions. These triggers are named `WSC_I_`T_i, `WSC_D_`T_i, and `WSC_U_`T_i, and its definition allows to intercept any write access (insert, delete or update respectively) to the T_i table, recording the event in the transaction report table (TrReport). The following example shows the definition of a basic WSC trigger, related to the insertion of a new object[2] into the table `MYTABLE`.

```
CREATE TRIGGER WSC_I_mytable
BEFORE INSERT ON mytable FOR EACH ROW EXECUTE
PROCEDURE tr_insert( mytable, getTrid(), NEW.l_mytable_oid);
```

Deletions and updates must also be intercepted by means of analogous triggers. However, as described above, the accessed fields can be optionally included in the transaction report (depending on the configuration of the MADIS middleware). To this end, a WSC trigger managing the updates should be *split* into a number of triggers, one for each field contained in the managed table `(WSC_U_mytable_field`**`1`**`, ...WSC_U_mytable_field`**`N`**`)`.

The second group of triggers is responsible for the transactions' readset collection. As already mentioned, this collection is optional, due to its high cost, and the fact that some consistency protocols can be accomplished without using readsets. To implement this collection, a view must be included for each table in order to compensate the lack of `TRIGGER ...BEFORE SELECT` in the SQL-99 standard. The original table must be renamed, and replaced by the new trigger. As views cannot be updated in several DBMSs, it becomes also necessary for the WSC triggers to be modified, in order to redirect the write accesses to the renamed original table. This can be done by implementing the WSC triggers as 'INSTEAD OF event' triggers, (in contrast to the basic `BEFORE event` detailed above). Finally, the `tr_insert`, `tr_update` and `tr_delete` procedures should be modified, in order to include the required redirection.

The last group of triggers added by MADIS is those responsible for the metadata management. In fact, this management can be disseminated in the WSC triggers detailed in this section. However, we describe here the metadata management implementation as independent triggers, in order to simplify the discussion. Whenever a new record is inserted, the DBMS must automatically insert the corresponding row in the metadata table. To this end, MADIS includes, for each table T_i, a trigger that inserts a row in the corresponding $MADIS\_Meta\_T_i$ table. As the `global_oid` is established based on the creator node identifier (i.e. the node where the object was created), and the lo-

[2] Note that the trigger executes the procedure `getTrid()` to obtain the transaction identifier.

cal object identifier in the creator node (managed in the $MADIS\_Global$ table), all fields contained in the $MADIS\_Meta\_T_i$ table can be filled without intervention of any consistency protocol.

Following the life-cycle of a row, when a row is accessed in write mode, the DBMS must intercept the access, and the metadata (e.g. version, timestamp, etc) of such object must be updated. To this end, a specialized metadata maintainer (MM) trigger is included for each table. The MM trigger updates the `version`, the `transaction identifier`, and `timestamp` of the record in the given metadata table. Finally, when an object is deleted, the corresponding metadata row must be also deleted. To this end, an additional trigger is also included for each table in the original schema.

Summarizing the tasks performed by the described triggers, it is easy to see that, for each table, only three triggers must be included: BEFORE INSERT, BEFORE UPDATE, BEFORE DELETE. Their implementation include both the transaction report management, and the metadata maintenance. If the readset management is a requirement, it is necessary to replace the definition of the triggers, implementing `INSTEAD OF` triggers, in contrast to `BEFORE` triggers. This allows the DBMS to redirect any write access to the adequate table, as well as to perform the metadata maintenance and the transaction management.

4 Consistency Manager

The architecture proposed by MADIS makes use of the database as the manager for most information related to consistency management. Moreover, the DBMS also provides the collected information to the consistency manager (CM) (situated on top of the database) with standardized structures.

Thus, the consistency management can be ported from a platform to another with a minimal effort. The rest of this section shows a Java implementation of a CM making use of the described schema modification.

Our Java implementation of the CM allows a pluggable consistency protocol to intercept any access to the underlying database, in order to coordinate both local accesses, and update propagation of committed local transactions (and, consequently, the local application of remotely committed transactions).

In our basic implementation of MADIS, we implement a JDBC driver that encapsulates an existing PostgreSQL driver, intercepting the requests performed by the user applications. The requests are transformed, and a new request is elaborated in order to obtain additional information (as metadata). The user perception of the result produced by the requests is also manipulated, in order to hide to the user applications the additionally recovered information. This mechanism allows the plugged replication protocol to be notified about any access performed by the application to the database, including query execution, row recovery, transaction termination requests (i.e. commit/rollback), etc. Thefore, the protocol has a chance to take specific actions during the transaction execution so as to accomplish its tasks.

Java user applications request a `MADIS Connection`, specifying the JDBC Driver to be used by the middleware to access the database. Query executions are also intercepted by MADIS encapsulating the `Statement` class. As response of user invocation

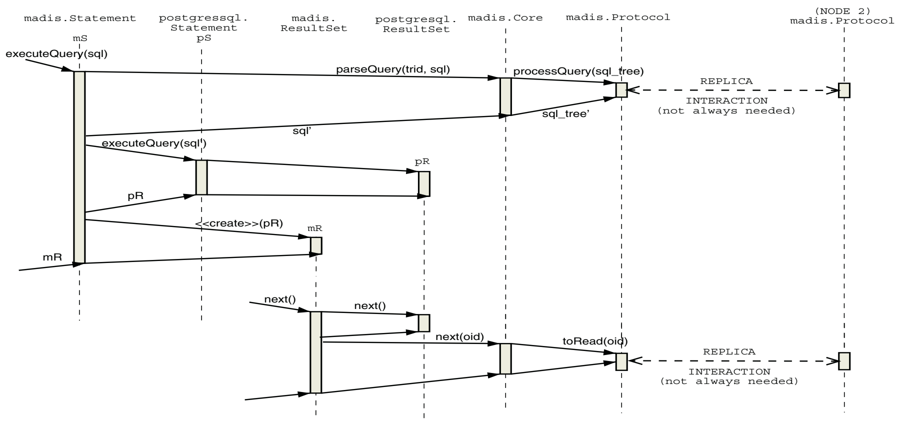


Fig. 1. Query Execution

to `createStatement` or `prepareStatement` the `MADIS Connection` generates `Statements` that manage user queries execution. When the user application requests a query execution, the request is sent to the `MADIS Core` class, which calls the `processStatement()` operation of the plugged consistency protocol.

Once this is done, the consistency protocol may modify the statement, adding to it the patches needed to retrieve some metadata, or collect additional information[3] into the transaction report. However, this statement modification is only needed by a few consistency protocols, which also have the opportunity to retrieve these metadata using additional sentences (on the "report-tables") once the original query has been completed. Optimistic consistency protocols do not need such metadata (like current object versions, or the latest update timestamps for each accessed object) until the transaction has requested its commit operation. So, they do not need these statement modifications on each query. The process for queries is depicted in figure 1.

Either if the application requests a *commit* as well as when a *rollback* is invoked, MADIS must intercept the invocation, and take additional actions. When the user application requests a commit operation, the `MADIS Connection` redirects the request to the `MADIS Core` instance. Then, the plugged protocol is notified, having then the chance to perform any action involving other nodes, access to the local database, etc. If the protocol concludes this activity with a positive result, then the transaction is suitable to commit in the local database, and the MADIS Core responds affirmatively to the `Connection` request. Finally, the `MADIS Connection` completes locally the commit, returning the completion to the user application after the notification to the MADIS Core. On the other hand, a negative result obtained from the protocol activity will be notified directly to the application, after the abortion of the local transaction. Finally, `rollback()` requests received from the user application must be also intercepted, redirected to the MADIS Core statement, and notified to the plugged protocol.

[3] The `ResultSet` should be also encapsulated in order to hide such included metadata.

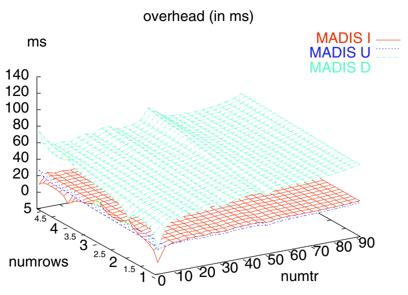
5 Experimental Results

As presented above, the proposed architecture is based on the modification of the database schema of an existing information system. With this technique, the database manager is the main responsible of generating and maintaining the information needed by any pluggable replication protocol to accomplish the tasks of consistency maintenance, concurrency control, and update propagation.

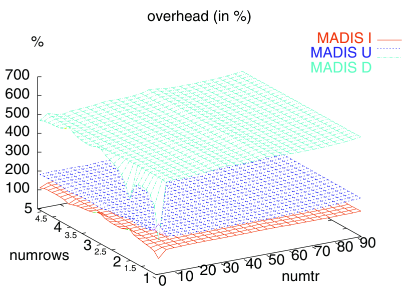
However, an important question to be discussed is the cost to be paid by the system from obtaining such benefits. This question, for our architecture, corresponds with the degree of performance degradation of the underlying database manager. Due to the overload introduced by the schema modification (i.e. triggers, procedures, added tables, etc) in the database, the database manager must deal with additional queries and this will redound in overheads from the common database functionality.

In spatial terms, the overhead introduced by the schema modification is easy to be determined, and leads out of the scope of this paper. Regarding computational overhead, our architecture introduces a number of additional SQL sentences and calculations for each access to the database when comparing with accesses to the original schema. Summarizing, *Insertion*, *Update* and *Deletion* operations need additional insertions on the `TrReport` table, and other operations with the corresponding `MADIS_Meta_`T_j table. In contrast *Selection* overhead varies depending on the plugged protocol. The readset collection may be performed in most of the cases by the middleware, just including the *local_*T_i*_oid* in the SQL sentences executed in the database. Thus, this inexpensive *oid* inclusion is often the overhead introduced in *Selection* operations. In this section, we discuss the overhead introduced in ***I**nsertion,* ***U**pdate* and ***D**eletion* operations, due to the relevance of the overhead in these operations. We are using a dummy consistency protocol, in order to calculate just the overhead introduced by the architecture.

The experiments consisted of the execution of a Java program, performing database accesses via JDBC. The schema used by the program contains four tables (`CUSTOMER`, `SUPPLIER`, `ARTICLE`, and `ORDER`). Each article references a row in the `SUPPLIER` table, and each `ORDER` references a `CUSTOMER` row, as well as an `ARTICLE` row. Each table contains additional fields as item description (a `varchar[30]`).



(a) `Absolute (ms) Overhead`



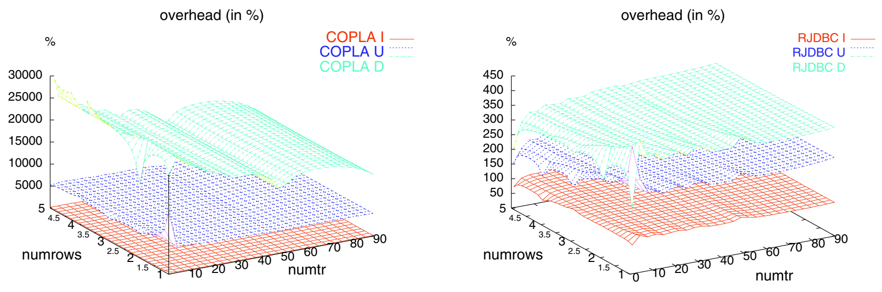
(b) `Relative (%) Cost`

Fig. 2. Mean Overhead

For each measurement, the experiment provides three values: the total cost of the *numtr* transactions of type I, U and D respectively, each one acting with *numrows* rows per table. We observed that deletions are the most overheaded operations in our core implementation. For a more accurate description of the overhead we calculated the time cost per transaction (figures 2 and 2(b)).

The results stabilized with a few number of transactions, which indicates that the system does not suffer appreciable performance degradation along the time. In addition, it is shown in figure 2 that the overhead per transaction is always lower than 80 ms in our experiments. Besides, figure 2(b) shows that the sensitivity for *numrows* is unappreciable (the system scales well in relation to managed rows) for any of the transaction types (I,U, and D). We concluded that our implementation of the MADIS database core introduces bounded overheads for Insertion and Update operations. However, Delete operations cause the schema modification to produce a dangerous, although bounded relative degradation of the performance (600% for 6000 rows deleted).

In GlobData, a middleware was developed to be used as a research tool in the field of replication protocols. In fact, several protocols were designed, developed, and implemented using this middleware. However, the architecture used in Globdata (COPLA) did not be conceived to provide low overheads in order to provide the required metadata to the plugged protocols. We include a comparison with COPLA. In the same conditions as the ones depicted in the previous subsections, we executed an equivalent test using COPLA. The conclusions (fig.3(a)) were that COPLA has a poor scalability for Update and Delete operations (50 and 200 times more costly than the standard schema).



(a) COPLA (b) RJDBC

Fig. 3. Relative Overhead

Finally, the MADIS architecture was compared with RJDBC. We consider RJDBC a lower bound of the achievable results in respect of metadata collection, although such approach doesn't scale well with regard to the number of connected nodes, due to the replication technique used (eager, pessimistic, and linear interaction). In RJDBC, there is no metadata maintained in the system. In contrast, all the requests to the database are just broadcast to any node in the system. When there is a unique node (as in our experiments), the system introduces a minimal overhead, consisting in the management of the requests. The experiments showed (figure 3(b)) that the system overhead remains stable proportionally to the number of rows processed. However, it is also shown that the overhead introduced for I and U operations is comparable to the introduced by MADIS.

6 Related Work

Considering the way the metadata collection is implemented, replication approaches can be classified as *Middleware-based* (where all the work is performed by a middleware external to the database), *Trigger-based* (where the collection is performed by triggers and callbacks to external procedures), *Shadow-Table-based* (using the shadow copies in order to build the update messages needed by other replicas), and *Control-Table-based* (based on timestamping of each row of the database). Each technique has its own benefits and drawbacks, as described in [9, 10]. There have been many implementations of middleware software providing database replication services.

In Postgres-R and Dragon [7], a DBMS core is modified in order to include distributed support to the database engine. This approach has a strong dependency on the database engine for which the system is developed, and it must be reviewed each time the original DBMS software release is updated. On the other hand, its performance is generally better than the one achievable using a middleware-based architecture.

In Globdata [4, 11], a middleware providing a standard API for Java applications was presented as a general solution for distributed database access. The system also included a heavy Relational-Objectual transformation. This allows the applications to make use of an object-oriented database schema, and the system translates this schema to a relational database. The system, although allows multiple consistency protocols to be plugged into, provides a propietary API for the applications to gain access to distributed databases, reducing the generality of the solution.

Also specific solutions for Java, implemented as a JDBC driver can be found in C-JDBC [12] and RJDBC [13]. The former emphasizes load balance issues, whilst the latter puts special attention to reliability. The implementation of these approaches are centered in Java, and porting the solution to other platforms has a high complexity, due to the characteristics of the specific techniques.

Finally, PeerDirect [9] uses a technique based on triggers and procedures to replicate a database. However, the system only includes one consistency protocol, providing particular guarantees, well fitted for a limited kind of applications.

7 Conclusions

Different applications require different kinds of managing replicated information. Hence, an adequate choice of appropriate replication protocols is due. Hence, a middleware which provides flexibile support for choosing, plugging in, operating and exchanging suitable protocols, including a homogeneous access to replicated databases, is desirable for many applications.

MADIS is a platform designed to provide such functionality. It supports an ample spectrum of diferent kinds of replication protocols. It is conceived as a two layers architecture. Most of the actual work is accomplished by the lower layer, which is implemented as part of an extension of the database schema. Its implementation makes use of standard SQL-99 database resources such as tables, views, triggers, constraints and stored procedures. Being independent of the underlying DBMS, its portability is easy and smooth. The lower layer consists of the collection of all information related to the accesses performed by the database transactions of a given application.

The upper layer makes use of this automatically collected information, by notifying the transactions' accesses to the currently plugged-in replication protocol. MADIS provides and allows to choose, plug in, run and perform on-the-fly exchanges of a wide range of different protocols, each one offering a particular choice of guarantees and behaviours to the user transactions. The implementation of this upper layer is simple enough to be ported from one platform to another with a minimal cost.

In this paper, we have described the MADIS lower layer, which is implemented as a set of SQL statements that modify the original database schema. As for the upper layer, we have exemplified the outlines of an implementation providing a Java JDBC standard API. This implementation enables a transparent, standard-conform access to replicated databases, without the need to make changes to the applications' code.

References

1. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
2. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada (1996) 173–182
3. Instituto Tecnológico de Informática: MADIS web site. *http://www.iti.es/madis* (2004)
4. L.Irún, F.Muñoz, H.Decker, J.M.Bernabéu: COPLA: A platform for eager and lazy replication in networked databases. In: 5th Int.Conf. Enterprise Information Systems. Volume 1. (2003) 273–278
5. Kemme, B.: Database Replication for Clusters of Workstations. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland (2000)
6. Agrawal, D., Alonso, G., El Abbadi, A., Stanoi, I.: Exploiting atomic broadcast in replicated databases. LNCS **1300** (1997) 496–503
7. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: Intl.Conference on Distributed Computing Systems. (1998) 156–163
8. Ferrandina, F., Meyer, T., Zicari, R.: Implementing lazy database updates for an object database system. In: Proceedings of the Twentieth International Conference on Very Large Databases, Santiago, Chile (1994) 261–272
9. PeerDirect.: Overview & comparison of data replication architectures (white paper) (2002)
10. Sybase, Inc.: Replication strategies: Data migration, distribution and synchronization. White paper (2003) 30 pages.
11. Rodrigues, L., Miranda, H., Almeida, R., Martins, J., Vicente, P.: The GlobData fault-tolerant replicated distributed object database. In: Proceedings of the First Eurasian Conference on Advances in Information and Communication Technology, Teheran, Iran (2002)
12. ObjectWeb: C-JDBC web site. Accessible in URL: *http://c-jdbc.objectweb.org* (2004)
13. Esparza-Peidro, J., Muñoz-Escoí, F.D., Irún-Briz, L., Bernabéu-Aubán, J.M.: RJDBC: A simple database replication engine. In: 6th Int.Conf.Enterprise Information Systems. (2004)